Concepts in Programming

An intermediate programmers tutorial for GFA BASIC 2.0 & GFA BASIC 3.0

Gottfried P. Engels

MichTron®

Concepts in Programming ™

An Intermediate Tutorial for all versions of GFA BASIC

For the Atari ST Series of Personal

Computers

Written by
Gottfried P. Engels

Edited by
George W. Miller

Published by MICHTRON, Inc.

576 South Telegraph
Pontiac, Mi. 48053

(313)334-5700

BBS: (313)332-5452

Concepts in Programming™

Published in the U.S.A. by MICHTRON, Inc.

576 South Telegraph
Pontiac. Michigan 48053

(313) 334-5700
BBS: (313)332-5452

Written by
Gottfried P. Engels

Edited by George W. Miller

English Translation by Inge Fitzsimmons

Cover Design by Thomas Logan

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

1988

ISBN 0-944-500-15-3

The opinions expressed in this book are solely those of the authors and are not necessarily those of MICHTRON, Inc.

Disclaimer of Warranty and Limits of Liability

The author and publisher of this book have used their best efforts in preparing this material and the programs contained in this book. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness.

The author and publisher make no warranty of any kind, expressed or implied, with regard to these programs or the documentation contained in this book.

The author and publisher shall not be liable for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.

Help!

Many people have difficulty typing listings correctly from published material. It's quite common for a user to type an incorrect number or letter, then not be able to identify the problem area when the program fails to function properly.

This problem is even more common with GFA BASIC listings. The absence of line numbers makes it even harder than normal to transfer a printed listing to your computer.

We suggest that if you encounter a problem with any program in this book, you follow several simple steps *before* you call us. First, check any data statements at the end of the program for a mistyped number. This is the most commonly encountered error.

Second, have another person read you the listing, line by line, from the book, while you check each program line on your monitor.

The code in each of the major programs is broken up into modules, with a brief explanation. Be sure you type each program in completely before trying to run it.

If, after carefully comparing the lines you have entered to the listings in the book, you still haven't found the problem call or write us. We may have additional suggestions for you.

We strongly recommend that you purchase the optional disk containing the major programs from this book. A special order form is located at the end of this book for your convenience.

MICHTRON, INC.

Table of Contents

Introduction

0. About this book

0.1 The Objectives

This book is intended for the average GFA BASIC programmer. The reader should be familiar with the command syntax of GFA BASIC. It would be helpful to have also written some small programs. Our goal is to assist users in the development of large programs in a structured manner.

Just how *DO* you develop large programs? Programs should be well structured so they can be easily corrected and extended with few errors during development. They should also run as fast as possible.

Many hobby programmers lack knowledge of the general principles of structured programming. A typical consequence is that many programmers make mistakes when writing a larger program. Hobby programmers like to read small, impressive "trick" literature. Knowing a few tricks is important, but more important is the fundamental knowledge which is often overlooked.

0.2 The Construction

The three sections of this book are not related. You can read each one without knowing the contents of the others.

The general principles of structured programming are explained in the first section. We tried to show, with examples, what consequences these rules have for GFA BASIC programmers.

This section also shows program optimization. Three areas are addressed which usually require the largest amount of time when a program is executed; loops, management of large data arrays, and communication with the disk drive or with other storage media.

Two larger programs are developed in the following sections. The programs are written in segments; each small segment is described and listed with comments and descriptive variable names. Each command is discussed as necessary. Suggestions are also made on how the programs can be further improved and expanded.

0.3 The Programs

The first program is a three dimensional drawing program. Complex three dimensional images can be constructed with this program and the finished objects may be rotated, moved, and resized. The figures can be represented as a wire-lattice model, or as solid objects. Instructions for output to nine and twenty-four pin printers are also included.

This program includes many interesting routines from GFA OBJECT, available from your ST dealer or you may order directly from MICHTRON, Inc.

The second program is a simple text editor. Extensive block functions, as well as the standard editing functions like delete, backspace, scrolling, loading and storing, searching and replacing, are available. Suggestions are made so you can turn this program into a complete text processor.

All programs in this book where written with GFA BASIC Version 2.0, and may be compiled using the GFA BASIC Compiler, Version 1.8 (or later versions). Users with later versions of GFA BASIC will also be able to use these programs, although Version 3.0 users should be aware that some optimization is available through the use of the command extensions available in GFA BASIC 3.0. An optional program disk may be ordered from MICHTRON, Inc. (See the coupon included at the back of this book.)

0.4 What will this book do for you?

What you get out of this book depends mainly on the time and effort you invest. It's possible to profit from simply reading it without trying any of the example programs on your computer. Of course, it's better to enter the programs and then modify them to suit your own specifications.

If you proceed step by step, you'll gain an understanding of structured programming, and you will also learn plenty of programming tricks.

The capabilities of all programs are limited by the programming language used. It's wrong to believe that the Atari ST is completely balanced with GFA BASIC. Each computer can only be totally balanced in assembler. However, in comparison to other programming languages on the ST, GFA BASIC is very fast.

GFA BASIC doesn't offer the flexibility of a language like C, but it is more uniform and easier to explore, especially for novice programmers.

I think that writing and testing programs is easier in GFA BASIC than in other languages. The Interpreter concept permits a comfortable interactive programming environment and the structured concept encourages a clear construction of the program itself. If GFA BASIC isn't fast enough for certain applications, time critical elements of these programs may be replaced with Assembly Language subroutines.

Structured Programming

1.0 Structured Programming and Program Optimization

This section deals mainly with two problems encountered during the program development; how to develop large programs which are clear and easy to correct, and how to write programs which run as fast as possible.

1.1. Structured Programming

The concept of structured programming has been mentioned quite often in connection with the BASIC language. It's been claimed that it's impossible to utilize structured programming in a language like BASIC. Typically, Pascal has been selected as teaching language, due to it's requirement of maintaining a proper program structure.

As time passed, structured programming became a centralized concept, especially in the educational field. In this section we are not going to repeat anything that has already been said about this thesis. Instead, we'll concentrate on the vital points of structured programming and what the advantages are for GFA BASIC programmers.

1.1.1 What is Structured Programming?

The effort to define structured programming has been made often, usually without results. You'll hear statements, like, "Structured programming is programming without using GOTO". Or meaningless statements, such as "Structured programming simplifies program development and modification." If this was the case, even a cup of coffee at 2:00 AM could be considered structured programming.

Unfortunately, there is no good definition of structured programming. With this in mind, here are some simple rules to follow.

- Programs should be clearly arranged.
- Correction of program errors should be easy.
- Modular construction of program segments should be possible.
- There should be few errors during the development process.

We'll examine a few of these rules, but first a few observations.

1. These are rules, not laws. They don't always have to be met. They serve only as guidelines.
2. These rules are especially important when:

   a) The program will be fairly large.

   b) The program contains very different parts, such as in integrated software packages.

   c) Several people develop different sections of a program.

   d) The source code is meant to be used by others and not just by the original author.

   e) A translation is planned into a different programming language or dialect of BASIC.

There are two common misconceptions about structured programming.

First, structured programming isn't a fantastic key for good programs and doesn't reduce programming to a mechanical process. The art of programming remains a creative process. Structured programming *does* help to eliminate non-creative problems which interfere with the creative programming process.

Secondly, structured programming isn't a new, revolutionary discovery. Only the strong emphasis on program structures, especially in the educational field, is new.

Structured programming simply is a way to produce relatively fast, good, flawless software. Try different techniques. Find out which one is best suited for your purposes.

1.1.2 Dividing of tasks and top-down programming

One of the basic principles of structured programming is the division of tasks. The total program must carry out a complex task. This complex task can be divided into many parts or functions. Each one of these smaller parts is carried out by a subroutine. The three most important derived principles concern the kind of subroutine, how the subroutine is activated, and its structure.

The top-down principle should be used to insure that subroutines work well together. This means that there is a main program routine having only a single task which calls other routines at the correct time. A typical example is:

```
DO
  ON MENU
LOOP
```

This routine calls a procedure which provides access to additional procedures. These procedures may in turn call other procedures which are even lower in the hierarchy. The following illustration of a top-down structure demonstrates a typical root construction.

Preparation

Main Program

loading — storing data
memorize search path

Here are a few simple rules:

1. A lower standing procedure in the hierarchy should never call a higher standing procedure with GOSUB or GOTO.

2. Each procedure should return to the procedure from which it was called after completing it's tasks.

Each program in GFA BASIC should start with preparation or initializing procedures in which arrays are dimensioned, some variables are declared, and (if necessary) pull-down menus are created. Then the main program follows.

The lowest step of the procedure hierarchy should be any error handling routines, such as ON ERROR GOSUB *procedure*. The installation of an error routine is a necessary precaution. Even in the most carefully tested program, errors still occur. Error handling routines should at least offer the chance to save any data to a disk file.

The sequence of writing the routines should be from top to bottom. To test higher program parts, such as menus, the lower program parts were developed as 'stubs'. A stub is a position keeper for a routine to be added later. It can be a short form of the actual routine or just consist of an entrance and exit (often referred to as "dummy" procedures).

```
PROCEDURE xyz
 '
 PRINT "XYZ"
 '
RETURN
```

This stub serves to make the menu testable before the total program is completed. The program stub prints it's name on the screen to signal that the correct menu branching is occurring.

Each subpart of a program should have an entrance and an exit point. Programmers familiar with other versions of BASIC often try to provide entrance and exit points for subroutines when switching to GFA BASIC. This usually results in an unstructured program construction. Program 1-1 represents a typical bad example. It's the menu of a small statistics program with data input and analysis subprograms.

Program 1-1. A Bad Example

Note! This is a BAD example!

```
'  This is an example for bad programming !
DIM x(10)
'
selection:
CLS
PRINT "What do you want: 1 = new input"
PRINT "                  2 = correcting data"
PRINT "                  3 = analyze data"
PRINT "                  4 = continue input"
PRINT "                  5 = replace existing
data"
INPUT w
'
' This is an example of bad programming !
'
' no ON GOTO in GFA BASIC
IF w=1
 GOTO data_input
ENDIF
IF w=2
 GOTO data_correction
ENDIF
IF w=3
 GOTO data_analysis
ENDIF
IF w=4
 GOTO continue_input
ENDIF
IF w=5
 GOTO replace_data
ENDIF
GOTO selection
```

```
'
' This is an example of bad programming !
'
data_input:
CLS
PRINT "New data name: ";
INPUT data$
replace_data:
n=0
continue_input:
FOR cnt=n+1 TO 10
 PRINT "No.";cnt;"";
 INPUT x(cnt)
 IF x(cnt)=0
  n=cnt
 ENDIF
 EXIT IF n=cnt
NEXT cnt
IF n=cnt
 DEC n
 GOTO data_analysis
ENDIF
n=10
PRINT "Do you still want to correct anything
(Y/N) ?"
INPUT a$
IF a$ <>"Y"
 GOTO data_ analysis
ENDIF
GOTO data_correction
'
data_correction:
CLS
PRINT "Positionkeeper Data correction, Push key"
VOID INP(2)
GOTO selection
```

```
'
' This is an example of bad programming !
'
data_analysis:
CLS
PRINT "Positionkeeper Data analysis, Push key"
PRINT "Name: ";data$
PRINT "N: ";n
FOR cnt=1 TO n
 PRINT "X";cnt;": ";x(cnt)
NEXT cnt
VOID INP(2)
GOTO selection
```

The data input routine has several entrance points; at *data_input*, *replace_data*, and *continue_input*. It also has several exit points; twice with GOTO *data_analysis*, and once with GOTO *data_correction*.

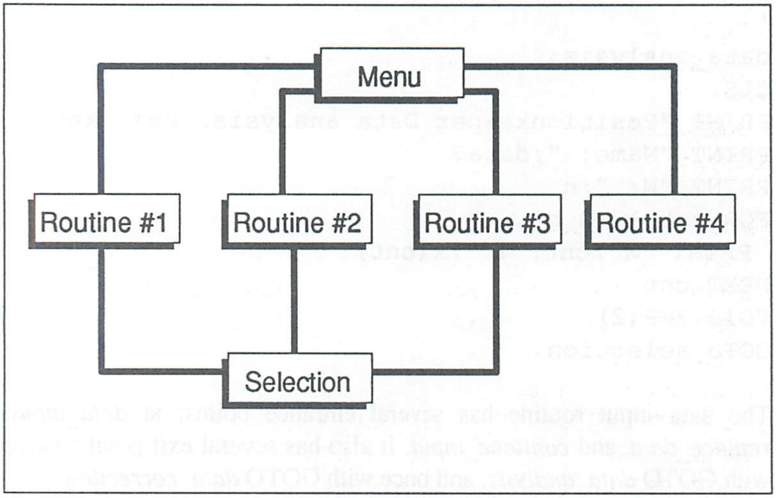
You really must try hard construct such a bad structure in GFA BASIC. The branching routine after the menu is very awkward, since GFA BASIC doesn't include an ON GOTO command.

Another example is the input loop. This program is supposed to exit the loop when a '0' is entered as input. In GFA BASIC, you can't jump out of a FOR NEXT Loop by using GOTO. The preceding example is really not all that rare. Programmers familiar with Pascal will adapt to GFA BASIC much faster than those familiar only with other versions of BASIC.

This example illustrates that it's possible to program in an unstructured manner in GFA BASIC. Structured programming is an attribute of the programmer, not the language.

Program 1-2 shows the structure of a task similar to the previous example. There is a triple step program hierarchy. On top is the menu. It calls the procedures for five menu points which form the second hierarchy step. Three of the procedures adjust parameters like *n* and *data$* and then call the input routine, which occupies the lowest hierarchy level.

Program Hierarchy



All procedures have only one entrance and exit, and always return to the part of the program which called it. In GFA BASIC they are implemented as shown by Program 1-2.

Program 1-2 A Better Method.

```
DIM x(10)
DO
 CLS
 PRINT "What do you want:     1 = new input"
 PRINT "                2 = correct data"
 PRINT "                3 = analyze data"
 PRINT "                4 = continue input"
 PRINT "                5 = replace available data"
 INPUT w
 ON w GOSUB input, correction, analysis,
continue, replace
LOOP
```

```
' _________________
PROCEDURE correction
 CLS
 PRINT "Positionkeeper Data correction, Push
key"
 VOID INP(2)
RETURN
'
PROCEDURE analysis
 CLS
 PRINT "Positionkeeper Data analysis, Push key"
 PRINT "Name: ";data$
 PRINT " N: ";n
 FOR cnt=1 to n
  PRINT "X";cnt;": ";x(cnt)
 NEXT cnt
 VOID INP(2)
RETURN
'
PROCEDURE continue
 CLS
 n=0
 @read_into_data
RETURN
'
PROCEDURE input
 CLS
 PRINT "New data name:";
 INPUT data$
 N=0
 @read_into_data
RETURN
'
```

```
PROCEDURE read_into_data
 FOR cnt=n+1 to 10
  PRINT "No.";cnt;"";
  INPUT x(cnt)
  EXIT IF x(cnt)=0
 NEXT cnt
 IF cnt=11
  n=10
 ELSE
  n=cnt-1
 ENDIF
RETURN
```

The construction of this listing isn't ideal either, but complies with the conditions of structured programming discussed thus far.

1.1.3 Program Planning

One result of structured programming is the timely occurrence of two functions; program planning and the actual writing of the program. Often, programmers spend time thinking about how to write the rest of the program instead of concentrating on the part of the program they are writing. Before you write even the first byte of source code, develop a complete plan for your program.

Put four sheets of paper aside for planning. On the first sheet of paper, describe what the program should do. On the second sheet, write down all the tricky problems which may be encountered.

The third and fourth sheets should contain information about program internals. On the third sheet write the variables you are going to use to accomplish the necessary functions. You only need to write down variables known to the whole program (global variables), not the assisting variables of each subroutine (local variables).

The variables listed would also include arrays with GEM information and maybe assisting arrays, which provide faster access to data. Variables such as maximum values and the actual number of data sets belong in this category, too. Only the variables listed on this sheet should be available to all routines in your program. They should be able to be changed by some of these routines. All other variables should be known only in the subroutine where they are used.

Sheet number four is based on the three previous sheets. Here you'll outline the structure of your program. Different program parts can be represented by small squares showing their functions, and connections of routines can be entered as lines. It's a kind of hierarchy diagram, not a flow chart or any other kind of diagram, representing the course of the program.

You'll probably throw these sheets away and start over several times. Don't immediately begin programming after outlining your plan. Think about the contents of the four pages. Don't think that you have to transform each idea immediately into program text.

Each sketch of a program shows strong individual preferences. No one can say that one solution is objectively better than another one. One good method is to design the general view of a program as hierarchy diagram and the single parts in form of a flow chart. Other programmers insist on Nassi Shneidermann diagrams. You'll have to decide which works best for you.

You'll probably need more time for the development of a program after changing to a systematic method of programming than with old unstructured methods.

Finding and removing errors (debugging) takes the most time. Don't expect to not make any mistakes while writing a large program. Besides, one of the main reasons for structured programming is to shorten the debugging phase.

The number of errors should be smaller during program development and the errors should be found much faster. You only need to check the value of a parameter at the entrances and exits of the procedures in question to find your mistake.

1.1.4 Routine Libraries

Structured programming simplifies finding errors and modifying programs.

Sorting algorithms are needed in many types of programs. It's nice to be able to set up a library of subroutines and merge the appropriate function without resorting to changing the program, or rewriting a function. The only requirement is that the routines don't interact with the main program. They must be able to run independently of the main program.

The easiest routines contain only a few commands and don't require variables to be passed to the routine or returned to the main program.

Local variables aren't required in the procedure demonstrated by Program 1-3. Reverse text, can be adjusted with two escape commands. In each case the command PRINT CHR$(27) is given, followed by a letter, "p" to enter text in the reverse video mode, or "q", to return to the normal mode. You may merge these two routines in programs when reverse text mode is needed.

Program 1-3 Reverse/Normal Text Mode

```
PROCEDURE revers_on
 PRINT CHR$(27)+"p";
RETURN
'
PROCEDURE revers_off
 PRINT CHR$(27)+"q";
RETURN
```

The way to call these routines could look like the example in Program 1-4.

Program 1-4 Calling Example

```
PRINT "test"
@revers_on
PRINT "test"
@revers_off
PRINT "test"
```

Such subprograms can consist only of one command, even though it rarely makes any sense. Computer programmers argue about the optimum maximum size of a subroutine. Some claim a subroutine may consist of up to 2000 command lines. My opinion is that it's not necessary to exceed more than 200 command lines, but this is only a rule of thumb. In general, it's best not to make the procedure too long, as this will complicate the debugging process.

Debugging becomes more complicated if routines contain variables, or submittal and return variables are used. All variables in subroutines should be declared as local variables so that other global variables used in the main, or calling program, can't be changed inadvertently when the subroutines are installed.

Passing values from a procedure to the main program does cause some problems. Imagine this situation: the FILESELECT command is used for selecting and loading data. Unfortunately, several things must be considered.

For example, you must check whether or not the cancel button or 'Ok' button has been selected. Also, it's inconvenient to check if the selected file name exists. The file select command should memorize the search path, because when you work with a hard disk you'll find it irritating to click through diverse files with each disk access, just to find your files.

It would be useful to write a routine that could be used instead of the file select command. It should be able to do what has already been stated, but now you need to consider the input and output parameters that will be used. The normal FILESELECT command knows three parameters: the search path, the default file, and the name of selected file. The path should be managed by the routine itself, so that the submittal parameter isn't needed when we ignore an extension to a file name like "*.DAT". The second parameter, the default file, is usually not needed.

The third parameter, the name of the selected file, is needed. It contains the name of the file selected and is the output parameter. To signal that everything functioned well, a kind of OK message must follow. This message can result by checking the file name returned. If loading isn't possible, an empty string may be returned.

How can a file name be submitted or returned? It would be awkward to read the file name in a variable that is used by the calling program. This makes it impossible to integrate the program module without changes into different programs. Instead, this routine should be useable if the following lines are included:

```
IF datafile$<>""
 OPEN "I",#1,datafile$
```

and also:

```
IF load_file$<>""
 OPEN "I",#1,load_file$
```

The routine must adjust this name. To make that possible, a pointer to this variable is passed instead of the variable itself. Program 1-5 demonstrates this concept.

Program 1-5 Pointer Example

```
a=0
@test(*a)
PRINT a
'
PROCEDURE test(value)
 *value=3
RETURN
```

Any name can be used instead of the variable name *value* or the variable name *a*, because the procedure is completely independent of the main program. It manipulates the variable, without the variable name being used in the procedure.

In this case, a string is passed and not a numerical variable. Program 1-6 is the complete listing of a routine to select a file to be loaded.

Program 1-6 Load File

```
PROCEDURE load_file(adr%)
 LOCAL filename$,cnt%
 '
 ' if the routine is called for the first time
 ' the path must be preadjusted.
 IF load_file.path$=""
  LET load_file.path$="\*.*"
 ENDIF
 '
 FILESELECT load_file.path$,"",filename$
 '
 ' if cancel has been selected clicked on or
 ' Ok button without fileselection.
 IF filename$="" OR RIGHT$(filename$)="\"
  '
  ' return empty string
  ' (shows, loading is impossible)
  filename$=""
 ELSE
 '
  ' determine path
  LET load_file.path$=filename$
  FOR cnt%=LEN(filename$) DOWNTO 1
   l%=LEN(load_file.path$)-1
   LET
load_file.path$=LEFT$(load_file.path$,l%)
   EXIT IF RIGHT$(load_file.path$)="\"
  NEXT cnt%
  LET load_file.path$=load_file.path$ + "*.*"
  '
  ' if file don't exist, return empty string
  IF NOT EXIST(filename$)
   filename$=""
   ENDIF
  ENDIF
 '
 ' execute return
 *adr%=filename$
RETURN
```

This procedure could be called as shown by Program 1-7.

Program 1-7 Call Load_file

```
datafile$=""
REPEAT
 @load_file(*datafile$)
 PRINT datafile$
UNTIL datafile$=""
```

In Program 1-6, there's a variable which isn't a local variable, *load_file.path$,* the variable which contains the path. The path can only be preserved between different calls to the procedure if the variable name is a global variable.

Now take another look at how variables are passed. In the main program (Program 1-7), a string isn't passed, instead, a pointer to the string is used.

```
@load_file(*datafile$)
```

The pointer contains a four byte storage address. In this case, the address of the string, or more specifically, the address of the first string descriptor byte.

```
*a$=ARRPTR(a$).
```

You'll learn in the sections of this book explaining program optimization what this is all about. The address can be held in an integer variable:

```
PROCEDURE load_file(adr%)
```

The value is returned through the assignment:

```
*adr%=filename$
```

This command means, take *a$* and put it onto the address of the variable *adr%*. It is completed internally, while the previously named string descriptor is being changed. More about this later.

1.1.5 Passing Parameters to Procedures

Earlier, we said that sub programs must be written so they can be universally used in your programs. If a sorting routine is needed, it should be possible to merge the routine and to pass the array to be sorted.

In GFA BASIC only single values may be passed to a procedure and not a whole field. As a consequence, it isn't possible to pass the array to be sorted. Therefore, in the sorting routine, the variable name of the sorting array is adjusted to the name of the actual arrays. This is contradictory to the principle of independence of the module from the total program, and is annoying work for the programmer. If you want to sort an array *x%()* and another time the array *y%()*, it would be necessary to write two routines.

This can be avoided if a pointer to the array is passed to the procedure instead of the array itself. This is accomplished by using the pointer character (*).

One advantage of GFA BASIC is that by using the command:

```
SWAP *pointer,field()
```

it's possible to address the array which the pointer points to with the name of the array *field()*. All desired operations are carried out with this name and the routine is finished by repeating the command. Program 1-8 is an example using a quick sort routine to sort an integer field.

Program 1-8 Quick Sort

```
DIM x%(25)
PRINT "Previous:"
FOR cnt%=1 to 20
 x%(cnt%)=RANDOM(100)
 PRINT x%(cnt%)
NEXT cnt%
'
@integer_quicksort(*x%(),20)
'
PRINT AT(40,1);"Afterward:"
FOR cnt%=1 to 20
 PRINT AT(40,cnt%+1):x%(cnt%)
NEXT cnt%
VOID INP(2)
'
'------------------------------------
'
PROCEDURE integer_quicksort(pointer%,n%)
 LOCAL cnt%,j%,border%
 SWAP *pointer%,sort%()
 @i_qsort(1,n%)
 SWAP *pointer%,sort%()
RETURN
'
PROCEDURE i_qsort(l_index%,r_index%)
 cnt%=l_index%
 j%=r_index%
 border%=sort%(INT((l_index%+r_index%)/2))
 REPEAT
  WHILE sort%(cnt%)<border%
   INC cnt%
  WEND
  WHILE border%<sort%(j%)
   DEC j%
  WEND
  IF cnt%<=j%
   SWAP sort%(cnt%),sort%(j%)
   INC cnt%
   DEC j%
  ENDIF
 UNTIL cnt%>j%
 IF l_index%<j%
  @i_qsort(l_index%,j%)
 ENDIF
 IF cnt%<r_index%
  @iqsort(cnt%,r_index%)
 ENDIF
RETURN
```

In the procedure *integer_quicksort*, the process is carried out, while the procedure *i_qsort* contains a normal quicksort working with the variable name *sort%()*.

The topic of structured programming is almost complete, but we would like to point out a few things about the two programs in sections two and three.

The processes described about structured programming were rules. They are not strongly observed in the two programs (3D Graphic and Text Editor), as local variables are not used.

Other items, such as top-down construction, are used. We already mentioned that the introduced principles are not carved in stone, but should be viewed as guide lines. It isn't really necessary to comply with these rules in both programs.

In conclusion, the two examples will show you what *NOT* to do in GFA BASIC. Even in GFA BASIC it's possible to program with bad structure.

1.1.6 Bad examples and GOTO

GFA BASIC could be viewed as a good instructional language for beginners. A lot of sins against structured programming are not even permissible in GFA BASIC. For instance, jumping into a procedure by using GOTO isn't allowed.

The command GOTO will always be mentioned when bad programming style is discussed. This command is the horror of all Computer Science instructors and one of the main reasons why BASIC has such a bad reputation.

Take a look at programs written in older versions of BASIC. In many programs wild jumping back and forth occurs, making it difficult to analyze the program. This type of programming is often referred to as spaghetti code. Mainly, using GOTO, without an IF statement, contributes a great deal to the lack of order in programs.

This doesn't mean that programs which use GOTO are badly structured. Structure isn't a sign of the utilized language or used commands; structure stems from the programmer. The only difference is that in some languages, it's easier to develop a clean structure.

Conservative basic dialects can be poison for beginners, while GFA BASIC can be very educational. Languages like Pascal and Modula II are very nice, but not ideal for beginners because of the compiler concept and their ponderousness.

Often a programmer depends on conservative languages to use GOTO. Here's an example in GFA BASIC. A FOR-NEXT loop can be exited with EXIT IF. If a loop is supposed to count to 1000, and there's no certain exit condition (here x(cnt)=0), the code could be written in GFA BASIC as follows.

```
FOR cnt=1 TO 1000
  ' commands
  EXIT IF x(cnt)=0
NEXT cnt
or
cnt=0
REPEAT
  INC cnt
  ' commands
UNTIL cnt=1000 OR x(cnt)=0
```

These commands weren't available in older BASIC dialects. However, they could have been written as:

```
10 FOR CNT=1 TO 1000
20 REM COMMANDS
30 IF X(CNT)=0 THEN GOTO 50:REM OUT OF LOOP
40 NEXT CNT
50 REM
```

As you can see, the structure of the three listings is identical. It wouldn't make sense to call the last one unstructured just because it contains a GOTO.

GOTO isn't needed at all in GFA BASIC, although this doesn't mean that all programs will automatically have a good structure. A meaningful case for using GOTO in GFA BASIC is shown in the following situation.

It's impossible in GFA BASIC to leave a procedure at different points by using RETURN. To simulate this, a label can be set before the RETURN command of the procedure. This label can then be jumped to from within the procedure by using GOTO.

The following two example programs will show you that even in GFA BASIC, it's possible to program unstructured and poorly. Both examples were found in the same edition of a large German computer magazine.

Program 1-9 shows needless interlocked IF-THEN conditions, a common error in GFA BASIC structure. We'll only give a structure, since the listing of the procedure is too long to be completely itemized.

The main thing in the procedure is to wait for a key press and then to branch to a certain procedure.

Program 1-9 Another Bad Example

```
PROCEDURE bad_example1
 key%=INP(2)
 IF key%=48
  GOSUB P1
  GOTO return
  ELSE
   IF key%=56
    GOSUB P2
    GOTO return
   ELSE
    IF key%=55
     GOSUB P3
     GOTO return
    ELSE
     IF key%=52
      GOSUB P4
      GOTO return
     ELSE
      IF key%=50
       GOSUB P5
       GOTO return
      ELSE
       '
       ' and so on n procedures
       ' for the appropriate key.
```

The procedure did end with:

```
               ENDIF
              ENDIF
             ENDIF
            ENDIF
           ENDIF
          ENDIF
         ENDIF
        ENDIF
       ENDIF
      ENDIF
     ENDIF
    ENDIF
   ENDIF
  ENDIF
 ENDIF
 return:
RETURN
```

A total of 15 ENDIF's followed by the label marking the jump point!

The GOTO's and the return jump mark aren't necessary because the ELSE commands lead to the same result. The interlocking of key polling leads to a very badly arranged structure, which is emphasized at the end with the long ENDIF chain.

Take a closer look at the program construction. The interlocking wasn't necessary. Two events are possible as soon as the program meets one of the IF THEN conditions. Either it was the wanted key, in which case the pertinent procedure is called, or it was a key not satisfying the condition. In that case the next following conditionals are tested.

Program 1-10 is slightly better, but still not perfect.

Program 1-10 A better way

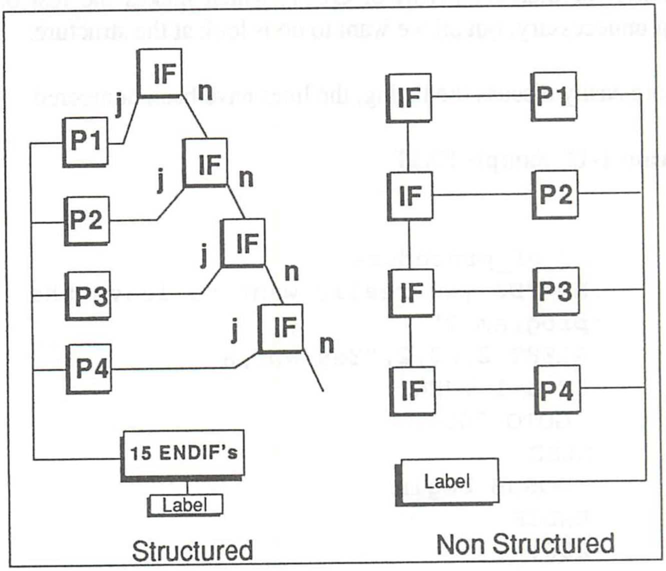
```
PROCEDURE not_interlocking
 key%=INP(2)
 '
 IF key%=48
  GOSUB P1
  GOTO return
 ENDIF
 '
 IF key%=56
  GOSUB P2
  GOTO return
 ENDIF
 '
 IF key%=55
  GOSUB P3
  GOTO return
 ENDIF
 '
 IF key%=52
  GOSUB P4
  GOTO return
 ENDIF
 '
 ' et cetera
 '
```

At the end of the procedure, the ENDIF-chain isn't needed; it only says:

```
 return:
RETURN
```

This construction with the many IF's is still not very elegant, but will work with GFA BASIC (Version 2.0), which doesn't have commands which are the equivalent of the switch/case command in C or the SELECT CASE of GFA BASIC Version 3.0.

Both program structures can be compared in the following illustration.



Program 1-11, the next bad example is even worse. The idea of this listing should be to exit a program running in the GFA BASIC interpreter, without a detour through the editor. Of course, there is a command for that, SYSTEM or QUIT, which makes the rest of the listing unnecessary, but all we want to do is look at the structure.

To more easily discuss the listing, the lines have been numbered.

Program 1-11 Sample EXIT

```
 1   end_of_procedure
 2     N$="Do you really want to leave the
       program ?"
 3     ALERT 2,n$,2,"Yes/No",a
 4     IF a=1 THEN
 5       GOTO 200
 6     ELSE
 7       GOSUB begin
 8     ENDIF
 9     200:
10   N$="Do you want to leave the interpreter
     too ?"
11   ALERT 2,n$,2,"Yes/No",a
12   IF a=1 THEN
13     SYSTEM
14   ELSE
15     END
16   ENDIF
17   END
18 RETURN
```

This procedure has several problems. It can't be made into a collection of subroutines because it will expect a *PROCEDURE begin* in the host program. This doesn't have to be available (line 7). If in *PROCEDURE begin*, RETURN is reached, the program jumps back into *PROCEDURE end*, which isn't really elegant.

The GOTO in line 5 uses a number as a label. This should be avoided since line numbers aren't used in GFA BASIC. If GOTO's are used, then all labels should have strong descriptive names. In this case, GOTO is completely unnecessary. If lines 5, 6 and 9 are eliminated, the function of the procedure will not change. The interpreter will then simply jump past ENDIF. Line 17 is unnecessary as well. The command END can never be reached. An END or SYSTEM command will be executed. The construction of this listing hasn't been thought through.

1.2 Optimizing Programs

This section is going to explain speed optimizing. Other optimization techniques, like length of program, and so forth, will not be discussed.

1.2.1 The Clean and Not So Clean Doctrine

Again, the principles of structured programming should only serve as guidelines for programs, not as absolute laws. In some situations, structured programming principles will not serve any useful purpose. Such situations occur when it is desirable for a program to run at it's maximum possible speed.

Your objection should be that you want to construct all programs to run as fast as possible, and structured programming doesn't measure up. In many instances speeding up a program to it's maximum isn't effective.

For example, in the non-interlocked version of the first two bad examples previously shown, GOTO's are found which jump to the end of the procedure. This is faster than if the routine would check all IF THEN conditions, even if the first condition is already fulfilled and all other conditions can only be wrong. But the time won is so small that GOTO's aren't worthwhile.

Especially drastic is the difference between structured programming and speed optimizing with the use of procedures and local variables. If you file a function in a subroutine, it will be slower than to construct the function as a procedure. It's even slower to use passed parameters and local variables. However, the loss of time is worthy of consideration only if these procedures are constantly called in a time critical loop.

Under normal circumstances, utilize the principles of structured programming. Deviations should be used only to achieve higher speeds of execution, or when structured programming make a program complicated.

One example would be if you use a sorting routine only once in a program. A sorting routine which is completely independent from the rest of the program as described above, would be more complicated than a less independent one.

In programming, three areas are potential causes for slowness. One problem area is loops, where commands are repeated several times. The second one concerns the management of large amounts of data in arrays. The third area is disk access. These three topics are going to be the center of attention in the following discussion about program optimization.

1.2.2 Loops

In order to optimize programs for speed, you'll need some information about the speed of different commands. In many cases there are various options available to solve this or that problem, and usually one of these options is much faster than the others. Up to now, the most discussed topic of this kind was probably the loops. The most important information about this topic will be repeated here.

In the following listing you'll find five possible ways for constructing a loop which counts from 1 to 500000. Four of them are the in GFA BASIC loop commands; FOR NEXT, REPEAT UNTIL, WHILE WEND, and the DO LOOP. The fifth loop has been constructed with the help of GOTO.

The times have been shown for a compiled program as well as a version run by the Interpreter. For more serious applications, only the compiler times should be of interest. Integer variables have always been used for variables, since they cause a program to execute faster than with real variables.

Program 1-12 shows the loop-benchmark program used.

Program 1-12 Loop Benchmark

```
' loop-benchmark  LOOP.LST
'
' FOR-NEXT loop
t%=TIMER
FOR cnt%=1 TO 500000
 ' commands
NEXT cnt%
PRINT "Time FOR-NEXT loop:"(TIMER-t%)/200
'
' REPEAT-UNTIL loop
t%=TIMER
cnt%=1
REPEAT
 ' commands
 INC cnt%
UNTIL cnt%>500000
PRINT "Time REPEAT-UNTIL loop:"(TIMER-t%)/200
'
' WHILE-WEND loop
t%=TIMER
cnt%=1
WHILE cnt%<=500000
 ' commands
 INC cnt%
WEND
PRINT "Time WHILE-WEND loop:",(TIMER-t%)/200
'
' DO loop
t%=TIMER
cnt%=1
DO
 ' commands
 INC cnt%
 EXIT IF cnt%=500000
LOOP
```

```
PRINT "Time DO-LOOP:",(TIMER-t%)/200
'
' GOTO-loop
t%=TIMER
cnt%=1
beginning_of_loop:
INC cnt%
' commands
IF cnt%>500000
 GOTO end_of_loop
ENDIF
GOTO beginning_of_loop
end_of_loop:
PRINT "Time GOTO-loop:",(TIMER-t%)/200
'
VOID INP(2)
```

The results of these loops in the interpreter and compiler are given in seconds. Running the program again would yield slightly different times. Such deviations are normal.

Type of loop	Interpreter	Compiler
FOR NEXT	29.61	8.52
REPEAT UNTIL	128.41	4.92
WHILE WEND	39.75	5.95
DO LOOP	138.33	5.69
GOTO	145.89	6.21

First look at the interpreter. The FOR NEXT loop is a definite winner. Since more serious programs are usually compiled, the second column is the more interesting.

The best looping structure in the compiled version is the REPEAT UNTIL loop, while the FOR NEXT loop, which performs so well in the interpreter, finishes last. Obviously, in time critical situations with compiled programs, you'll want to use the REPEAT-UNTIL structure.

Now, don't take out all your old programs and change the loops. If you are using many slow commands within a loop, changing the loop will help very little.

Constructions such as:

```
FOR cnt%=1 TO 10000
 x%(cnt%)=cnt%
NEXT cnt%
```

should be changed if they are used frequently.

In loops such as:

```
FOR c%%=1 TO 10000
 x%(c%)=SIN(c%)+COS(c%)+TAN(c%)+ATN(c%)
NEXT c%
```

the percentage of speed gained is marginal, since trigonometrical commands demand much calculation time and are very slow.

1.2.3 Array construction in GFA BASIC

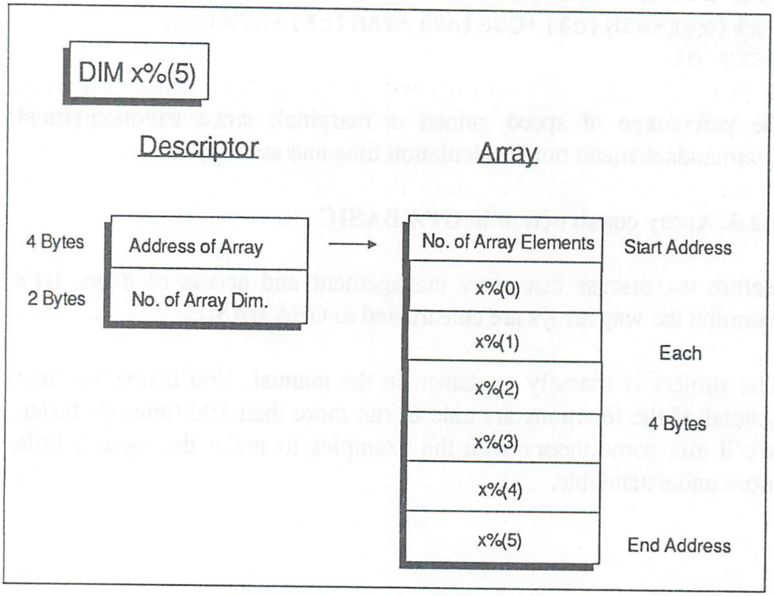
Before we discuss fast array management and access of disks, let's examine the way arrays are constructed in GFA BASIC.

The subject is sparsely explained in the manual. You'll now see that several of the functions are able to run more than 100 times (!) faster. We'll mix some theory with the examples to make the topic a little more understandable.

1.2.3. Numerical Arrays

Let's start with a numerical array, the easiest one. The command DIM x%(5) allocates space for a six element array of integer (whole) numbers. Since each of these six values occupies a four byte storage area, 24 bytes (6*4) have been made available for data. An array of real numbers, DIM x(5), will allocate a storage area of 36 bytes. (Each item in the array requires 6 bytes. 6*6=36 bytes.)

For the time being, we'll stay with the integer array, X%(). The 24 data bytes are aligned one behind the other in storage, and not in a jumble. The following illustration shows the position of values in memory and the structures which manage the array.



Each of the 4 bytes contains the value of *x%()*. Values with lower index numbers are in lower addresses than values with higher indexes. In memory immediately before *x%(0)* a four byte value which indicates how many elements the array can hold. The address of this value is the beginning address of the array.

Also, there is an additional six byte long structure containing more information about the array. This structure is called the *array-descriptor*. The first four bytes of this descriptor contains the address of the first byte in the array. The remaining two bytes state how many dimensions are in the array. The value must be at least one.

The address of the descriptor can be determined by using the function ARRPTR. *ARRPTR(x%())* would return the address of the first byte of the descriptor. Before discussing details about the structure of other types of arrays and multiple dimensional arrays, let's look at an example building on what we've learned so far.

If you have two one dimensional arrays of the same size, i.e. *x%(20000)* and *y%(20000)*, and the data contained in array *x%()* needs to be manipulated, but the original data must still be available. This means that the contents of array *x%()* must be filed elsewhere. The easiest way is to copy array *x%()* into array *y%()*.

Many programs use the slow method of element-wise copying, for copying arrays. Program 1-12 demonstrates one typical slow copy method

Program 1-13 Slow Array Copy.

```
DIM x%(20000),y%(20000)
FOR cnt%=0 TO 20000
 y%(cnt%)=x%(cnt%)
NEXT cnt%
```

It can be done much faster by using the BMOVE command. The speed of these two different methods is compared in this table:

	FOR loop	BMOVE	Speed Factor
Interpreter	50.10	0.635	78.9
Compiler	11.92	0.625	19.0

The commands for array copying have been executed ten times in a FOR NEXT loop to make the message dependable. You can see that the BMOVE method is incredibly fast, even if it can't be speeded up with the compiler.

Let's look more closely at using BMOVE. The BMOVE command moves a copy of an area of memory, starting at address *from%* that is *number%* bytes long to the location specified by *to%*. The proper command syntax is:

```
BMOVE from%, to%, number%
```

Two storage areas have been defined to integrate the data of arrays *x%()* and *y%()*. To copy one array into another, the storage area of data of one array should be copied into the storage area of the other array. The BMOVE command does this, but think about the specifications of the parameters for BMOVE. First, we need the address of the starting the point for the original arrays.

This location can be found by using *ARRPTR(x%())* to get the address of the descriptor of the array. *LPEEK(ARRPTR(x%())* results in the start of the array. (This address holds the 4 bytes containing the starting address of the field.) The first four bytes at this address state only how many subdivisions *X%()* has. This information is insignificant to us. The beginning of the meaningful data is then:

```
LPEEK(ARRPTR(x%()))+4
```

The address to which the data bytes are to be copied, can be determined the same way.

```
LPEEK(ARRPTR(y%()))+4
```

Alternatively, the command VARPTR() can be used. VARPTR returns the address of a specified variable.

For example:

```
LPEEK(ARRPTR(x%()))+4=VARPTR(x%(0))
```

All that's needed now is the number of bytes to be moved. Since a total of 20001 (starting from zero) items of data are in the array, and each item needs 4 bytes, 80004 (4*20001) bytes must be copied. The copying command results in:

```
BMOVE LPEEK(ARRPTR(x%()))+4,
LPEEK(ARRPTR(y%()))+4,80004
```

or even:

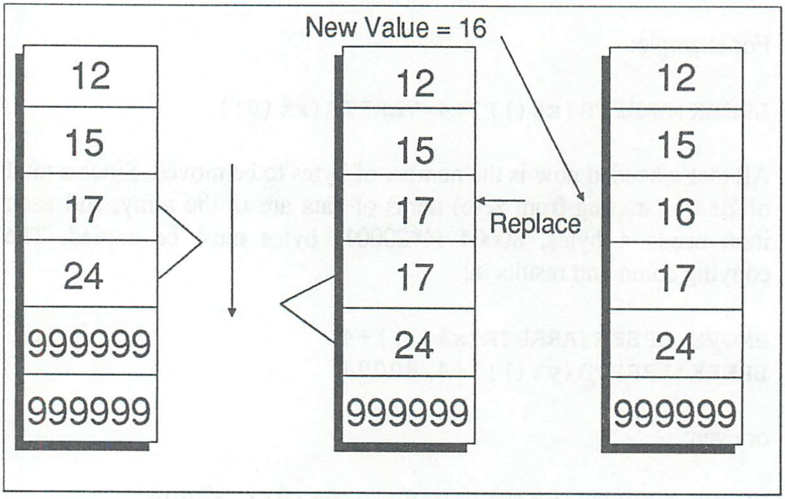
```
BMOVE VARPTR(x%(0)),VARPTR(y%(0)),80004
```

This looks more complicated than a loop, but the increase in speed is worth the effort, especially with multiple dimensional arrays, which will be discussed later.

Here are two typical application examples. Suppose you need a routine which reads numbers into an array and files each newly given number by it's size correctly into the array.

Such a routine could get a number from a user with the INPUT statement, then search for the first number in the list which is larger than the new number, looping from 0 to the most recently read number. Much faster search routines are available (for example, the binary search).

In the array of numbers, all elements from the number which was larger than the new one could be moved up by 4 bytes (equal to one value). In the now vacant position, the new number can be written. The drawing on the next page shows this principle.



This idea can be represented by Program 1-14.

Program 1-14 Sorting.

```
' put in numbers and sort immediately
SORTREIN.LST
'
DIM number%(20)
ARRAYFILL number%(),999999
addr%=LPEEK(ARRPTR(number%()))+4
'
REPEAT
  ' read new value in
  PRINT AT(1.1);"New value";number%+1;":";
  INPUT new%
  CLS
```

```
 '
 ' look for value, from where will be inserted.
 ' here just a primitive procedure.
 FOR cnt%=0 TO number%
  EXIT IF number%(cnt%)>new%
 NEXT cnt%
 ' insert value
 IF number%>cnt%
  BMOVE addr%+cnt%*4,addr%+cnt%*4+4,(number%-
cnt%)*4
 ENDIF
 INC number%
 number%(cnt%)=new%
 '
 ' give out list
 FOR cnt%=0 to number%-1
  PRINT AT(40,cnt%+1);number%(cnt%)
 NEXT cnt%
 '
UNTIL number%=20
'
PRINT "More than 20 elements are not
permissible."
VOID INP(2)
```

The practical value of Program 1-14 is rather limited, but it does demonstrate the possibilities of for the BMOVE command. Now try to determine the BMOVE parameter with VARPTR().

First the array is dimensioned for the numbers and each item is filled with a large value. The large numbers are the cancellation criterion for the loop looking for the value filed before the new value.

Next, the address is calculated from where the data are. In the following loop, the new value (*new%)* is obtained by using the INPUT statement. Then FOR NEXT loop is used to look for the first value in the array which is larger than the input value, *new%.*

After the search, the variable *cnt%* is in the position of the first value which is larger than *new%*. All elements of array, *number%()*, will be moved starting from this position.

VARPTR or a pointer may be used to get to the address of a variable in array.

The two expressions:

```
LPEEK(ARRPTR(x%()))+4
```

and

```
VARPTR(x%(0))
```

are identical.

You can place lines on the screen without the using graphic commands. In Program 1-15, an array is constructed where the starting address for screen storage (found by using XBIOS(3)) is used. The array is as large as the screen storage area.

Data is written into array, and onto the screen through a loop by using the ARRAYFILL command. The result is a series of slowly wandering lines. Again, the practical use of this program is limited, but does provide a useful demonstration.

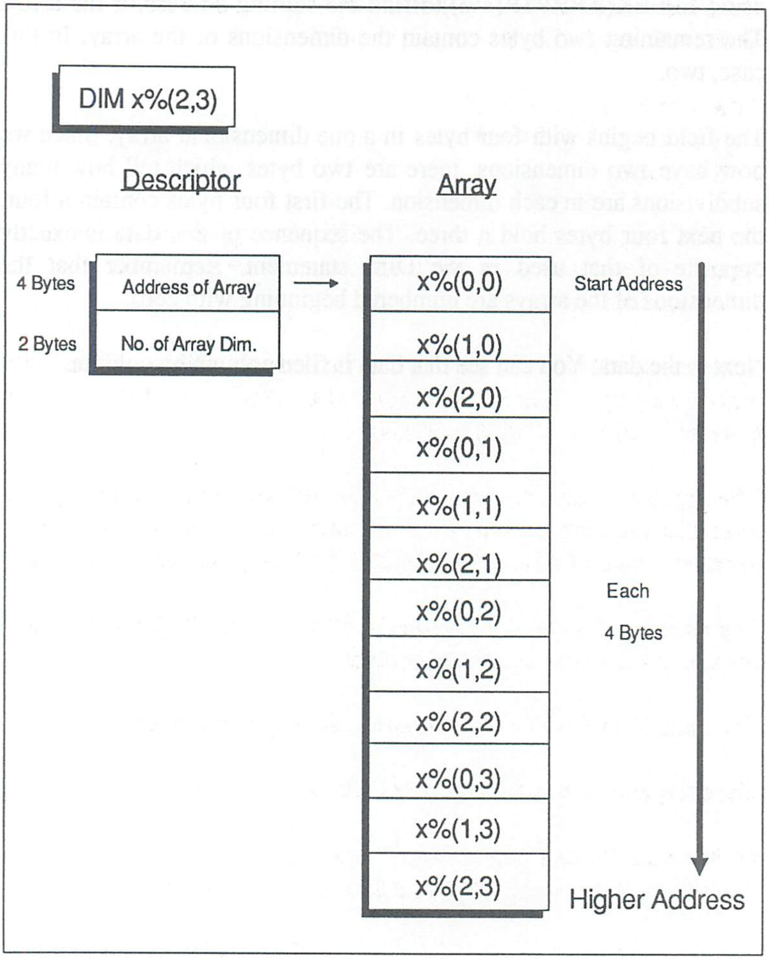
Program 1-15 ARRAYFILL demo.

```
'
DIM x%(0)                    ! reserve space
for array
LPOKE ARRPTR(x%()),XBIOS(3)-4 ! address of array
LPOKE XBIOS(3)-4,8000         ! size of array
'
DO
 FOR cnt%=0 TO 30
  ARRAYFILL x%(),2^cnt%
 NEXT cnt%
LOOP
```

Take a look at page 42. The address of the array descriptor is determined by ARRPTR(x%()). The first four bytes can be read by using LPEEK(ARRPTR(x%))to find the starting address of the array. The remaining two bytes contain the dimensions of the array. In this case, two.

The field begins with four bytes in a one dimensional array. Since we now have two dimensions, there are two bytes which tell how many subdivisions are in each dimension. The first four bytes contain a four, the next four bytes hold a three. The sequence of this data is exactly opposite of that used in the DIM statement. Remember that the dimensions of the arrays are numbered beginning with zero.

Next is the data. You can see that data is filed column by column.


DIM x%(2,3)
Descriptor
Array
4 Bytes
Address of Array
2 Bytes
No. of Array Dim.
x%(0,0)
Start Address
x%(1,0)
x%(2,0)
x%(0,1)
x%(1,1)
x%(2,1)
Each
4 Bytes
x%(0,2)
x%(1,2)
x%(2,2)
x%(0,3)
x%(1,3)
x%(2,3)
Higher Address

How can you use this information? Just imagine you have dimensioned an array as x%(20000,1). You want to use this array to frequently copy the value with an index of one into variables with an index of zero. The usual method of doing this is shown below.

```
FOR cnt%=0 TO 20000
 x%(cnt%,0)=x%(cnt%,1)
NEXT cnt%
```

As with copying, you may use BMOVE. Now try to use the preceding figure to determine what parameters should be used with BMOVE before reading on.

The starting address for BMOVE is the starting address of field plus eight (skipping the array dimension information) plus the number of bytes to be moved (20001*4=80004).

The destination address, where the copy of these bytes is going to be stored, is the starting address of the data bytes. This address may be found by using LPEEK(ARRPTR(x%()))+8, as previously explained.

The number of bytes to be moved is 80004. VARPTR() could be used instead of the descriptor for the address.

Try using VARPTR() in the following listing just for practice.

The complete command with ARRPTR()is:

```
address%=LPEEK(ARRPTR(x%()))+8
BMOVE address%+80004,address%,80004
```

The time gained using BMOVE is significant when compared to a FOR-NEXT loop, as shown by the example on the next page.

	LOOP	BMOVE	Acceleration factor
Interpreter	8.90	10.05	178
Compiler	2.17	0.05	43

For your own experiments with BMOVE, try Program 1-16.

Program 1-16 Two dimensional array.

```
' TWO_DIM.LST
DIM x%(10,1)
address%=LPEEK(ARRPTR(x%()))+8
'
FOR cnt%=0 TO 10
 x%(cnt%,1)=cnt%+100
NEXT cnt%
'
PRINT "Before:"
FOR cnt%=0 TO 10
 PRINT x%(cnt%,1),x%(cnt%,0)
NEXT cnt%
'
BMOVE address%+44,address%,44
'
PRINT "After:"
FOR cnt%=0 TO 10
 PRINT x%(cnt%,1),x%(cnt%,0)
NEXT cnt%
VOID INP(2)
```

Here's a problem for you to solve. An array is defined as DIM x%(20). The first 11 (0 to 10) elements are the numbers 0 to 10. In the second array, (DIM y%(10)) consists of the numbers 11 through 20. Move the Y-array into the X-array so that the numbers 0 to 20 are in X-array.

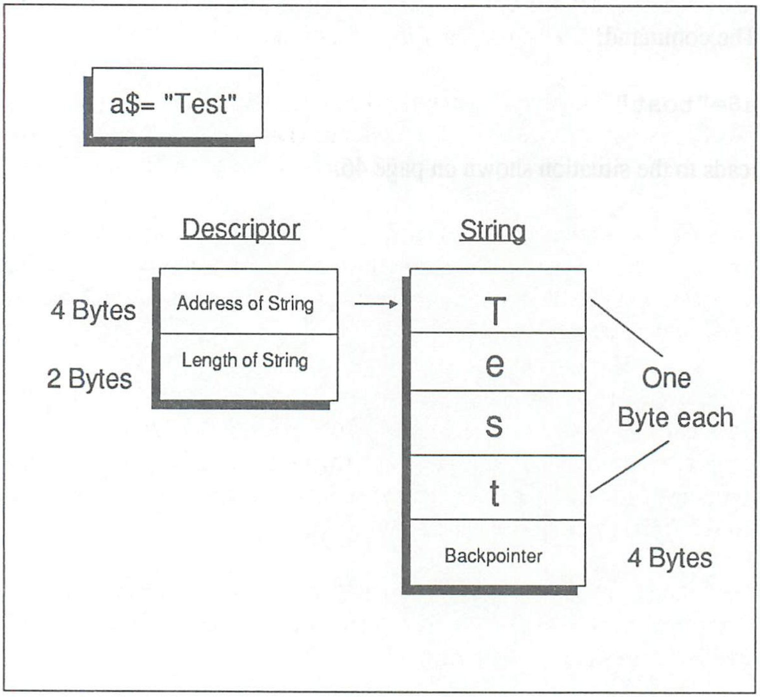
1.2.3.2 Character String Arrays

Let's examine how a single string is stored in memory by GFA BASIC.

The command:

```
a$="test"
```

leads to the situation shown on page 46.


a$= "Test"
Descriptor
String
4 Bytes
Address of String
2 Bytes
Length of String
T
e
s
t
One
Byte each
Backpointer
4 Bytes

A string descriptor is assigned to the string (a$). The first four bytes of the descriptor contain the address of the string. The next four bytes contain the length of the string, in this case, 4. The starting address of the string is determined by using LPEEK(ARRPTR(a$)). You could also use VARPTR(a$) or a pointer operation (*a$).

The starting address contains the first character of the string. Successive bytes contain the remaining characters, one byte for each character. The string is padded with zero's, if it has an uneven length.

A four byte long value following the string must end on an even address. These four bytes contain the address of string descriptor. This construction is called a back pointer; it's not necessary to take a closer look at the meaning right now.

The example used to illustrate this grey topic (Program 1-17) bothers me a little. I'm using a trick which you should not, under any circumstances, use in one of your programs. A string without a back pointer plays an important role, and such a construction can make a fast move to the reset button necessary.

The theme of the example is pretty expressive; changing the text notices of the GFA BASIC Interpreter and manipulating menu INPUTs and messages from the desktop.

Program 1-17 is shown with line numbers for ease of discussion.

Program 1-17 CHANGE.LST

```
   1 ' change of BASIC notices and desktop
INPUTs CHANGE.LST
   2 area$=""
   3 s_from%=BASEPAGE+30000
   4 REM s_from%=75000   ! desktop-menubar
   5 '
   6 LPOKE ARRPTR(area$,s_from%
   7 DPOKE ARRPTR(area$)+4,32760
   8 '
   9 DO
  10  PRINT "Which text should be changed:";
```

```
 INPUT search$
 EXIT IF search$=""
 '
 IF INSTR(area$,search$)
   PRINT "Ok, entry found.  New text: ? ";
   FORM INPUT LEN(search$),new$
   from%=s_from%+INSTR(area$,search$)-1
   BMOVE VARPTR(new$),from%,LEN(new$)
 ELSE
   PRINT "Sorry, did not find text."
 ENDIF
LOOP
```

Before reading the following discussion, type in Program 1-17. (Be sure to omit the line numbers, as GFA BASIC doesn't use them.)

Answer the question about which text should be changed with *'End of Program'*. The program should find the text and notify you, then ask what the new text should be.

Reply *'Finished'*. When the question about changing text appears again, press the <Enter> key to end the program.

A standard alert box will announce the end of the program, but this time the text in the alert box reads *"Finished"*. You have successfully changed the text.

Now, run the program again. This time replace *'Direct'* with *'Immediately'*. After returning to the editor, you'll see that the text has been changed on the command bar of the Editor screen.

Place a REM before the line which reads:

```
search_from%=BASEPAGE+32000
```

and remove REM from the line which reads:

```
search_from%=75000
```

With this program, you should now be able to make changes to the desktop. (This may not work for you, depending on which version of the operating system is installed on your ST.)

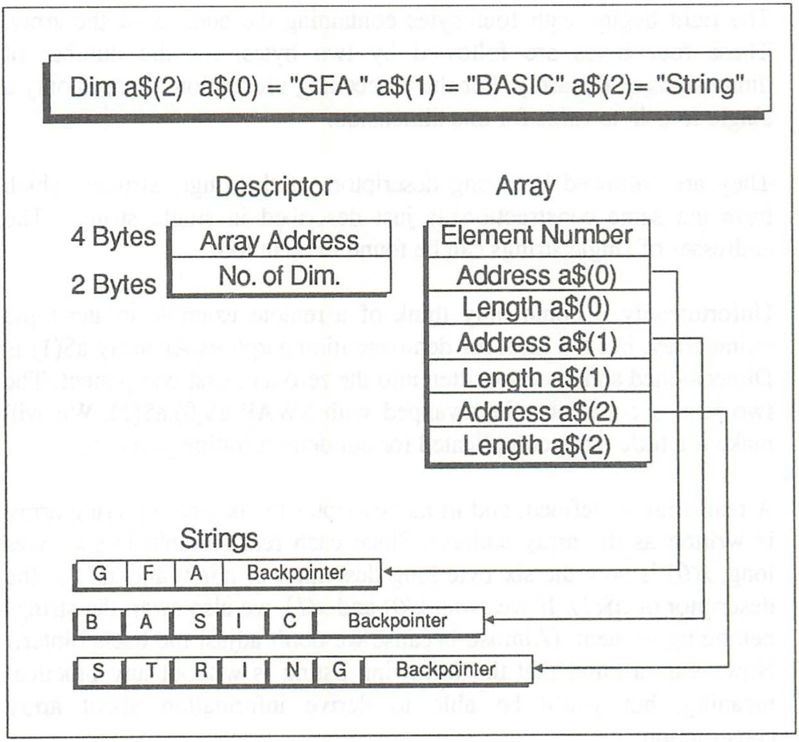
Run the program and enter as the search text *'Options'* and as replacement text *'BASIC'*. Exit the program, then return to the GEM desktop. If you were successful, the Options menu header has been changed. Other menu items and the text of desktop dialog and alert boxes can be changed.

From an understanding of the way string variables are stored, it's only a small step to understanding string arrays. These arrays also have a descriptor, an array address and the number of array dimensions.

Enter Program 1-18 into your computer, and run it. This program is the basis for the information contained on page 51 and the following discussion.

Program 1-18 Memory Check

```
DIM a$(1)
a$(0)="GFA"
a$(1)="BASIC"
addr%=ARRPTR(a$())
PRINT "Address of Array Descriptor: ";addr%
PRINT    "Number    of    elements    in    Array:
";DPEEK(addr%+4)
PRINT
PRINT "Variable"; TAB(10); "Address"; TAB(20);
"Byte"; TAB(30);"Character"
FOR x%=0 TO 1
 cnt%=0
 PRINT
"A$(";x%;")";"=";CHR$(34);a$(x%);CHR$(34)
 addr%=VARPTR(a$(x%))
 WHILE PEEK(addr%)<>0
  PRINT    TAB(10);    addr%;"    ";    TAB(21);
  PEEK(addr%);" ";TAB(35); CHR$(PEEK(addr%))
  INC addr%
 WEND
 pad:
 IF PEEK(addr%)=0
  PRINT   TAB(10);     addr%;"    ";   TAB(21);
  PEEK(addr%);" ";TAB(35); CHR$(PEEK(addr%))
  INC addr%
  GOTO pad
 ENDIF
NEXT x%
PRINT
PRINT "Waiting for a key press..."
VOID INP(2)
```


Dim a$(2) a$(0) = "GFA" a$(1) = "BASIC" a$(2)= "String"
Descriptor
Array
4 Bytes
2 Bytes
Array Address
No. of Dim.
Element Number
Address a$(0)
Length a$(0)
Address a$(1)
Length a$(1)
Address a$(2)
Length a$(2)
Strings
G F A Backpointer
B A S I C Backpointer
S T R I N G Backpointer

By using ARRPTR(a$()), we can determine the address of the descriptor.

The field begins with four bytes containing the address of the array. These four bytes are followed by two bytes, for the number of dimensions of the array. For the preceeding illustration, there is only a single four-byte value for one dimension.

They are followed by string descriptors of the single strings, which have the same construction as just described in single strings. The addresses of single strings can be found in them too.

Unfortunately, I could only think of a remote example to the topic string array, but it's ideal for demonstration purposes An array a$(1) is Dimensioned and text is written into the zero and first component. The two strings could now be swapped with SWAP a$(0),a$(1). We will make it a little more complicated for our demonstration purpose.

A real array is defined, and in its descriptor the beginning string array is written as the array address. Since each real variable is six bytes long, *x(0)* is now the six byte long descriptor of *a$(0)* and *x(1)* is the descriptor of *a$(1)*. If we swap *x(0)* and *x(1)*, we also swap the strings belonging to them. (Almost, because we don't adjust the backpointer.) Now bear in mind that the following listing is without any practical meaning, but you'll be able to derive information about array construction:

Program 1-19 String Swap

```
' just for fun.(incomplete) swapping of strings
for demo purposes
DIM a$(1)
DIM x(1)
a$(0)="test string"
a$(1)="text for testing"
LPOKE ARRPTR(x()),LPEEK(ARRPTR(a$()))
'
PRINT "before :";a$(0),a$(1)
SWAP x(0),x(1)
PRINT "after: ";a$(0),a$(1)
```

1.2.4 Communication with the Disk

Now we will get into the third announced theme of program optimizing, the loading and storing on floppy disks, hard disk or RAM disk. Often, long waiting periods are involved because the disk operating system (DOS) of the ST is not very fast.

1.2.4.1 Numerical Files

The downloading of larger data amounts usually means that arrays have to be saved and then loaded again later on. Usually a LOOP is used and data is either written or read element-wise. With what we now know about arrays, the suspicion is raised that this could go a lot faster. Since numerical arrays use continuous storage areas, arrays could be saved and loaded again by simply sending these storage areas to the disk and then picking them up again.

Saving of storage areas can be done with BSAVE or BPUT. BPUT will be written into a file in this case. This method has the advantage that several arrays can go into a file. The analog loading commands are BLOAD and BGET. The parameters of these commands are:

Address: storage address, it's loaded to this address or it is stored from this address

Number: number of bytes to be loaded or stored

BLOAD *file name, address*

BSAVE *file name, address, number*

BGET *#channel number, address, number*

BPUT *#channel number, address, number*

In the following listing, both loading methods are used. A real array is loaded and stored. In the first case the values in the array are not affected by commas. The second case is affected by commas. This also has an affect on element wise methods of storage area requirements. Files are transferred with:

```
PRINT x(cnt%);CHR$(10);
```

CHR$(10) (linefeed) separates files. This separation requires only one byte, while PRINT x(cnt%) uses two bytes. The storage area requirement for element wise storage is, for numbers without commas, two bytes (character number and separation code). For 10000 values this comes to 20000 bytes. The storage with BPUT needs six bytes for each array, or triple what the others require, that is: 60000 bytes. If the storage is done element wise with one position in front of the decimal point and ten post-decimal positions the following happens:

1	byte for predecimal position
1	byte for the decimal point; CHR$(46)
10	bytes for after decimal positions
+1	byte for separation code; CHR$(10)

13 bytes per value are needed, a total of 130000 bytes.

Since this method of transferring of storage area is faster, element wise storing only makes sense in a few exceptional cases to save storage area on the disk. Here's the listing with which loading and storing time has been tested:

Program 1-20 SAVELOAD.LST

```
' Benchmark for loading- and storing methods
SAVELOAD.LST
'
DIM x(10000)
'
PRINT "Integer Values:"
PRINT
FOR cnt%=1 to 10000
 x(cnt%)=RANDOM(6)+3
NEXT cnt%
@benchmark
'
PRINT
PRINT "Real numbers:"
PRINT
FOR cnt%=1 to 10000
 x(cnt%)=RANDOM(6)+Pi
NEXT cnt%
@benchmark
Void INP(2)
'
'
PROCEDURE benchmark
 ' Element wise storing
 t%=TIMER
 Open "O",#1,"test1"
 FOR cnt%=1 to 10000
  PRINT #1,x(cnt%);CHR$(10);
 NEXT cnt%
 CLOSE #1
 PRINT "Elementwise storing:",(TIMER-t%)/200
 '
 ' Elementwise loading
 t%=TIMER
 OPEN "I",#1,"test1"
 FOR cnt%=1 to 10000
  INPUT #1,x(cnt%)
 NEXT cnt%
 CLOSE #1
 PRINT "Element wise loading:",(TIMER-t%)/200
 '
 ' store with BPUT
 t%=TIMER
 OPEN "O",#1,"test2"
 BPUT #1,VARPTR(x(1)),60000
 CLOSE #1
 PRINT "Store with BPUT:"(TIMER-t%)/200
 '
 ' Load with BGET
 t%=TIMER
 OPEN "I",#1,"test2"
 BGET #1,VARPTR(x(1)),60000
 CLOSE #1
 PRINT "Load with BGET:",,(TIMER-T%)/200
RETURN
```

The times measured by this program are shown in the following table. If you want to repeat those measurements, you might find a discrepancy between your measurements and those shown on the table. There are several reasons; one is the number of files already on the disk. Also, the times were measured with a compiled program.

Times for Floppy disk:

	Element wise	Storage Area	Acc. Factor
Storing Integer Values	52.44	15.785	3.3
Loading Integer Value	24.73	5.595	4.2
Storing Real Values	170.025	15.83	10.7
Loading Real Values	158.865	5.595	28.4

Times on hard disk:

	Element wise	Storage Area	Acc. Factor
Storing Integer Values	29.045	5.62	5.2
Loading Integer Values	20.71	0.345	60.0
Storing Real Values	42.71	5.79	7.3
Loading Real Values	128.335	0.35	366.7 (!)

Times on RAM disk:

	Element wise	Storage Area	Acc. Factor
Storing Integer Values	24.7	1.095	22.6
Loading Integer Values	19.705	0.17	115.9 (!)
Storing Real Values	28.835	1.26	22.9
Loading Real Values	119.73	0.18	665.2 (!)

The acceleration factors depend strongly on several factors. A few items in the table are startling. The loading times of element referring methods are enormous, even from a RAM disk.

Equally high is acceleration factor. The speed difference between a hard disk and RAM disk depends mostly on the type of RAM disk used. The one used for these tests was not optimal in this regard.

1.2.4.2 Guidelines for Writing Programs

You can do a few favors for the user of your programs when accessing the disk. For example, you should determine the last adjusted search path and readjust it with the next file select call, so the user doesn't have to go through several directories constantly.

If you use the storage area method, it's easy to determine how long the files is going to be before storing it. Check that sufficient space is available on the disk and give the user an appropriate message.

If your program uses a certain data format, it may lead to terrible effects if the user mistakenly selects a file with the wrong format. Your program can be written in a manner that will check for the correct format before loading. Use a few bytes for identification in the files you create so that your loading routine will be able to recognize the file format.

Program 1-21 RECOG.LST;

```
' RECOGNIZE.LST
DIM x(10)
marking%=12345
'
OPEN "O",#1, "recognize.dat"
BPUT #1,VARPTR(marking%),4
BPUT #1,VARPTR(x(0)),66
CLOSE #1
'
FILESELECT"\*.*","",filename$
who_there%=0
'
OPEN "I",#1,filename$
BGET #1,VARPTR(who_there%),4
IF who_there%=12345
 ALERT 1,"Known Format.",1,"Return",a%
 BGET #1,VARPTR(x(0)),66
ELSE
 ALERT 1,"Unknown Format !",1,"Return ",a%
ENDIF
```

```
CLOSE #1
```

Try to select any file type other than those created with the proper ID, for instance, GFABASIC.PRG. It will be rejected with the comment “unknown format”. Two additional things should be considered while loading.

Check whether or not a file actually is on the disk before you try to open it.

Warn the user during a save routine if a file with the name selected already exists. You don’t have to pay attention to all those principles in programs for your own use, but programs you want to pass on should use these ideas.

1.2.4.3 Text Files

Up to now, we have only loaded numerical arrays, but now let’s look at loading text, i.e. ASCII files. If you have done this before with INPUT or LINE INPUT, you’re about to learn something new.

Suppose you wanted to write a program that removes all comments from a GFA BASIC listing. For that purpose the LST-file, the ASCII-version of the program, should be loaded and processed.

First we will develop the loading routine. Each line will be placed in a string of array Z$(). The conventional loading method would be:

```
DIM z$(5000)
OPEN "I",#1,"test.lst"
WHILE NOT EOF(#1)
 INC cnt%
 LINE INPUT #1,z$(cnt%)
WEND
CLOSE #1
```

Loading must be done with LINE INPUT instead of with INPUT, because INPUT interprets commas as row separations. Later, you'll learn that this method, even when loading from a RAM disk, is not very fast. An alternative would be to load storage areas again, for example with INPUT$, which has the following syntax:

```
z$=INPUT$(number_bytes,#channel number)
```

This command loads only a specified number of bytes without checking for the end of a row of text. It's necessary to search for the end of line character, the Carriage Return (CHR$(13)) and/or Linefeed (CHR$(10)). Here is a program listing to compare the speeds of the two methods.

```
' ASC_LOAD.LST
DIM z$(5000)
FILESELECT "\*.*","",filename$
'
' conventional loading method
T%=Timer
OPEN "I",#1,filename$
cnt%=0
While Not (Eof(#1))
 Inc cnt%
 LINE INPUT #1,z$(cnt%)
Wend
CLOSE #1
PRINT "conventional:";(Timer-T%)/200
'
' with INPUT$
t%=Timer
number%=256
OPEN "I",#1,filename$
connecting_passages%=Lof(#1)/number%
rest%=Lof(#1) Mod number%
row%=0
'
FOR cnt%=1 TO connecting_passages%
 Data$=INPUT$(number%,#1)
 @Analysis
```

```
NEXT cnt%
data%=INPUT$(rest%,#1)
@analysis
'
CLOSE #1
PRINT "With INPUT$: ";(Timer-T%)/200
'
Void INP(2)
PRINT "Number Rows: "row%
FOR cnt%=1 to row %
 PRINT z$(cnt%)
NEXT cnt%
PRINT "Ok"
Void INP(2)
' ---------------
PROCEDURE analysis
 l_string$=l_string$+data$
 REPEAT
  position%=INSTR(l_string$,CHR$(10))
  IF position%
   INC Row%
   z$(row%)=LEFT$(l_string$,position%-2)
   l_string$=MID$(l_string$,position%+1)
  ENDIF
 UNTIL position%=0
RETURN
```

As an example file for the measurements, I used a file with 1585 rows and about 43 kilobytes length. The times of the compiled test program were:

	LINE INPUT	INPUT$	Acceleration factor
Disk	106.30	19.65	5.4
hard disk	89.43	4.40	20.3
RAM disk	87.12	3.04	28.7

The loading time for RAM disk again depends a great deal on the type of RAM disk. As you can see, the more complicated looking loading method, using INPUT$ is vastly superior in terms of speed. The speed can be drastically changed by selecting the variable, Number%, the number of the bytes read with INPUT$. Here are a few more test results:

Number%	Time (hard disk))
10	9.74
256	4.40
10000	26.55

As you can see, the three potential "time-eaters"; loops, management of larger arrays, and communication with disk can, with some knowledge, be improved dramatically. Don't forget that a skillful selected data structure and good algorithms are at least as important as any kind of trick.

1.2.5 Getting Fancy with the Atari ST

The Atari ST has one operating system (TOS) and one user interface (GEM). They don't represent the most modern and fastest systems, but support the programmer excellently, offering a number of useful routines which can be easily included in your own programs.

If you are interested in developing good programs in GFA BASIC, you must become familiar with the routines in TOS and GEM. I only want to mention it here, because there are already many books and articles about this subject. One representation of these routines from the view of GFA BASIC can be found in the *GFA BASIC Book* by Frank Ostrowski. Other information is also available in the *GFA BASIC Programmers Reference Guide, Vol. I*, by George Miller.

3-D Graphics

2. Development of a 3D Graphic Program

For quite some time, the world of computer graphics has been enthralled with three dimensional graphics. Many articles can be found on this subject in magazines and books. Also more and more 3D pictures can be seen on TV and in movies, all created by a computer.

2.1 3D Graphics on the ST

Even though the ST offers plenty of technical performance, strangely enough, only a few 3D graphic programs are available. Comparable computers like the Mac and the Amiga have a plentiful supply. There are a few books and magazines which deal with 3D graphics on the ST, but they have not shown up in many programs.

One such program is GFA OBJECT, written completely in GFA BASIC. Many routines could be used in a program developed from GFA OBJECT, which have been simplified and supplied with other variable names.

Unfortunately, the ST isn't as efficient as the graphics computers which develop the figures we admire on TV. However, it is strong enough to provide an entry into the basics of 3D graphics.

2.2 What should the 3D Program be able to do?

The program we will now develop will serve to create and manipulate (almost) all three-dimensional elements (relatively) easily. The manipulation should include turning, moving and changing dimensions. The building of objects can be easily carried out in the form of a unitized construction.

Simple basic figures are put together with the mouse. These basic figures can be created with the help of generators for rotations and translation elements.

The kind of storage of objects in the computer determines which operations can be carried out with them. We will talk about those different data structures in the next section.

The finished objects should be viewed from different perspectives. The easiest way is to simply draw lines which connect the corner points of the object. This representation is called "wire model", because the sides of the object look only like wire with no invisible planes hanging between them. All sides are visible; even those which are supposed to be covered by planes.

The next step is to make a diagram of the invisible sides. This is called a "Hidden-Line" process. A similar term is "Hidden-Surface". The two processes are often confused. Hidden-Line only determines which lines should be totally or partly covered, with Hidden-Surface the complete plane is determined.

Hidden-Line-algorithms are out of fashion now, because they did not allow such operations as "Ray-Tracing" and "Texture-Mapping", which make an object appear real. Ray-tracing is a procedure that exams the characteristics of a ray of light. This is necessary for creating reflections, images and similarities. Texture-Mapping means to project a texture, a surface structure or pattern, onto an object. It is almost impossible to make a distinction between an image generated with these technics and a photograph. Our program will not contain these procedures.

We will stay at the threshold of Hidden-Surface by using an algorithm which is so simple that we can almost avoid talking about Hidden-Surface. But the results are sufficient for our purposes. It connects with a hatching-algorithm, which is very capable despite it's simplicity. Unfortunately, it stays insignificant on the ST with filling-pattern-hatching and the minimum amount of colors (without special programming tricks) at the same time.

I can not advance far enough in this section with the description of the basics of 3D Graphics because extensive mathematical basics are required, and that alone would fill another book. If you are interested, you should read one of the many books about this subject. A classic is:

Newman, W.M., Sproull, R.F.:

Principles of interactive computer graphics,

McGraw-Hill, 1979

This is an old book, but it's still worth reading.

2.3 Problems associated with programming 3D Graphics

These programs don't contain especially difficult programming tasks in comparison to the text editor in the next chapter. But we should think about a few problems associated with maintaining a clean program structure.

The following questions are especially important:

1. How do we keep the objects in storage, and what does the data structure which allows management and manipulation of data look like ?

2. What mathematical basics are behind the object manipulations?

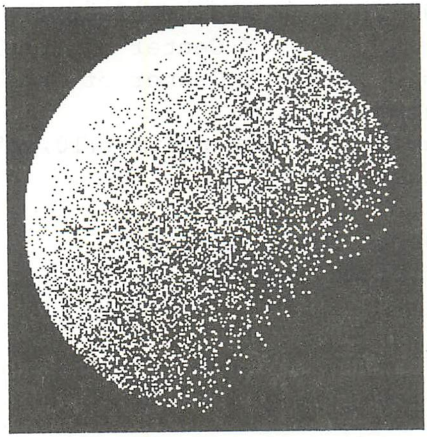
3. How can the program be written so that it will run in all screen resolutions, not only on a monochrome monitor or only on a color monitor?

2.3.1 Search for a Data Structure

Several possible data structures are available which allow management of three dimensional objects in the computer and displaying them on the screen. Here are two examples which offer some idea of how different these structures can be.

One representation shows that objects are put together with simple basic figures with the help of a linear equation. The size, position, and form is determined by the parameter of equation.

This example demonstrates this concept with a sphere. The following illustration was created with this program.



Program 2-1 SPHERE.LST

```
' SPHERE.LST
'
GRAPHMODE 3                 ! Replace Mode
radius%=100                 ! Radius of Sphere
xm%=320                     ! X-Center of Sphere
ym%=200                     ! Y-Center of Sphere
'  Black Background
PBOX xm%+radius%+10,ym%+radius%+10,xm%-radius%-
10,ym%-radius%-10
'
'  Direction of Light Rays and
' length of light vector
lx=-1.5
ly=-1.5
lz=-0.25
ll=SQR(lx^2+ly^2+lz^2)
'
' Loop over all points of sphere surface
FOR x%=xm%-radius% TO xm%+radius%
 FOR y%=ym%-radius% TO ym%+radius%
  '
  '  In case points are within Sphere radius
  IF SQR((x%-xm%)^2+(y%-ym%)^2)<radius%
   '
   vx%=x%-xm%
   vy%=y%-ym%
   vz%=SQR(radius%^2-vx%^2-vy%^2)
   vl=SQR(vx%^2+vy%^2+vz%^2)
   co_angle=(lx*vx%+ly*vy%+lz*vz%)/(ll*vl)
   IF co_angle+RND(1)*1.5-0.75>0
    PLOT x%,y%
   ENDIF
   '
  ENDIF
  '
 NEXT y%
NEXT x%
VOID INP(2)
```

2.3.2 The Mathematical Basics

Now we must deal with some basic mathematics, a necessary evil. We'll try to represent everything as simply as possible and restrict our discussion to the absolute minimum. There will be no long explanations about WHY's, only short explanations about HOW's.

Trigonometric and vector calculations are necessary when discussing functions such as rotating an object and changing the size of an object. Let's begin with the easiest one, moving an object.

Each point of an object is described through it's X-, Y- and Z-coordinates. To move an object in reference to the points, the same transformation must be done at each coordinate point. All that is necessary is to add the value by which the object is to be moved to the coordinate of the appropriate dimension.

This can be accomplished with the following loop if *X(cnt)* is the X-coordinate of the *cnt* point and the entire object is to be moved with all points along the X-axis.

```
FOR cnt%=1 TO n
 ADD x(cnt%),20
NEXT cnt%
```

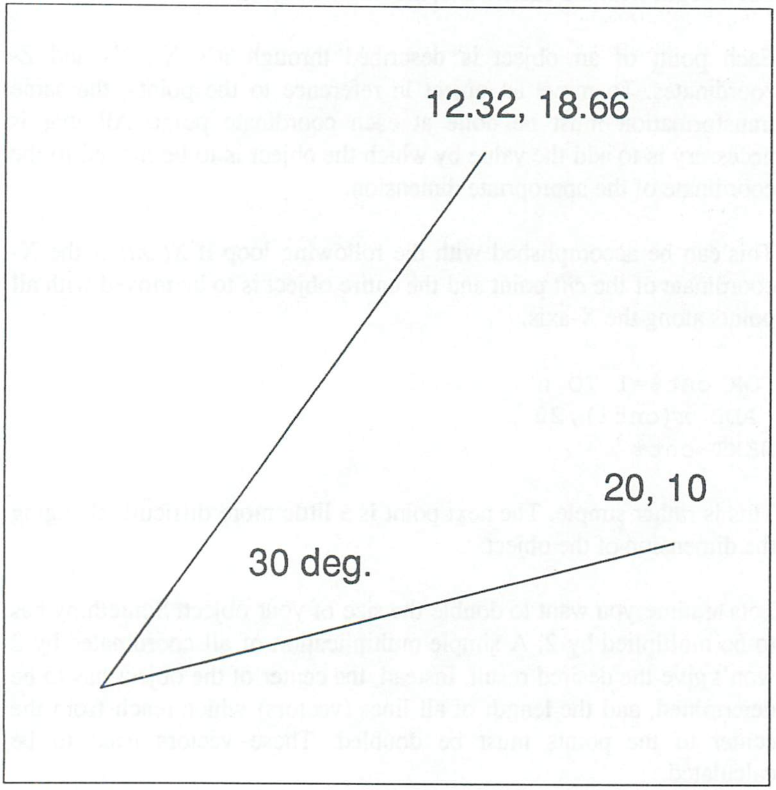
This is rather simple. The next point is a little more difficult; changing the dimension of the object.

Lets assume you want to double the size of your object. Something has to be multiplied by 2. A simple multiplication of all coordinates by 2 won't give the desired result. Instead, the center of the object has to be determined, and the length of all lines (vectors) which reach from the center to the points must be doubled. These vectors need to be calculated.

The new vectors are obtained by simple subtraction of the point coordinates and center coordinates. Then, the new values of these vectors are multiplied by 2 and the resulting values are added to the center coordinates.

In other words: The object is moved by subtraction of center coordinates from the zero point, blown up and moved back into the center.

Somewhat more difficult is the rotation of a point by one turn center. Take a look at the following sketch:



You see a point with the coordinates (x1,y1)=(20,10). It is rotated around the zero point by a rotation angle *w*, ending up at point (x2,y2). The calculation may be done with these formulas:

```
x2 = x1 * COS(w) - y1 * SIN(w)

y2 = x1 * SIN(w) + y1 * COS(w)
```

If you'd like to know why, I would suggest you refer to a good mathematics book, as we don't have space here. We are only concerned with programming. We know that the SIN and COS functions of GFA BASIC expect the angle in 'arc' measurements and not in degree. The conversion may be done with this formula:

```
Arc = Degree * Pi/180
```

Furthermore, attention has to be paid that these formulas rotate one point around the zero point of the coordinate cross. What we are doing is making a new point the center of the object being rotated. If the center of the object is (Xm,Ym), before the calculation, the values would be:

```
X1 = X1-Xm
X1 = y1-Ym
and after:
X2 = X2+Xm
Y2 = Y2+Ym
```

Let's calculate the rotation for the simpler case of rotation around the zero point. The coordinates were (*x1*,*y1*) = (20,10), the rotation angle is 30 degrees:

```
dw=30*PI/180      ! rotation angle
x2=20*COS(dw)-10*SIN(dw)
y2=20*SIN(dw)+10*COS(dw)
```

This results in: x2=12.32 and y2=18.66; the rotation occurs in the mathematically positive direction, which is counterclockwise.

This was only a rotation in two-dimensions. What is needed for three-dimensional rotations?

We can simplify things here. With each rotation around one axis in three-dimensions, one coordinate of the three coordinates remains unchanged. This is the coordinate around which the object is rotating.

Our two-dimensional rotation can be understood as rotation about the screen reaching the Z-axis, while the Z-coordinate remains the same. In fact, it doesn't even enter into our calculations. According to that, the other rotations in the three-dimensional area are:

Around X-axis:

```
y2 = y1 * COS(w) - z1 * SIN(w)
z2 = y1 * SIN(w) + z1 * COS(w)
```

and around Y-axis:

```
x2 = x1 * COS(w) - z1 * SIN(w)
z2 = x1 * SIN(w) + z1 * COS(w)
```

That covers the basics for object manipulation. The remaining problem is that of shading. The necessary method was used in the last section of the listing for the sphere.

If the light lights up some areas of an object more than others, shadings of the object occur. The direction of the light rays and the orientation of the area play a major role.

If the light hits a vertical plane perpendicularly, it is lit to its maximum intensity. If it hits the edge of the plane, the light won't hit the surface because the rays are running parallel to the surface..

The light intensity results from the relationship of the angle of the light rays and the plane surface. A vector, standing vertically on a plane, can be easily calculated if you know three points of the plane (none of which is on the vector). We can use three corner points with the coordinates (x1,y1,z1), (x2,y2,z2) as well as (x3,y3,z3). The vertical standing vector on the plane is described through the coordinates (vx, vy, vz) which are calculated with the following formulas:

```
vx=(y2-y1)*(z3-z1)-(z2-z1)*(y3-y1)
vy=(z2-z1)*(x3-x1)-(x2-x1)*(z3-z1)
vz=(x2-x1)*(y3-y1)-(y2-y1)*(x3-x1)
```

We are able to determine from what direction the light rays are coming. This direction is determined by the vector (Lx,Ly,Lz). The equation

```
Co_angle=(Lx*vx+Ly*vy+Lz*vz)/(Ll*Lv)
```

results in the Cosine of the angle, which is surrounded by vectors. Ll is the length of the light vector, Lv is the length of the vertical standing vector on the plane.

The cosine usually provides information about the color to be selected by using the formula:

```
Color number=(Co_angle+1)*(number_color/2)
```

The colors should be in order by their degree of brightness (intensity). The transitions between colors can become smeared with a coincident number. In the sample equation, the problem has been solved somewhat differently, because it is an extreme case where the shading can only be done in black and white.

If you want to take a deeper look into 3D Graphics, you must delve much more deeply into mathematical problems. The best way would be to find a book which shows the mathematical basics from the viewpoint of a graphic-programmer.

Ordinary math books can also be used, but they usually don't contain much of the material necessary for computer graphics.

2.3.3 Adjusting of a Program for all Screen Resolutions

When writing a graphics program it's sometimes helpful to keep in mind the specific screen resolution under which the program is intended to run. Then you needn't be concerned about the appearance of the screen in other resolutions.

Our 3D program will run in all resolutions. To achieve this, each graphic output is provided with a correction factor which makes the adjustment to the actual screen resolution. The possible resolutions (640x400, 640x200 and 320x200) have the pleasant quality of dividing the number of picture points during inspection of resolution into equal halves. You do this by bisecting either the Y-coordinates or the X-and Y-coordinates.

The basic idea behind adjusting resolution is to refer all outputs to the high resolution and to divide the Y-coordinates by 2 at the middle resolution. All coordinates have to be divided by 2 in the lower resolution.

Therefore, at the beginning of the program the resolution is questioned; this is done with XBIOS(4), and two correction factors are established after the result of this function. These are:

In the higher resolution: (640x400): xk%=1 and yk%=1

In the middle resolution: (640x200): xk%=1 and yk%=2

In the lower resolution: (320x200): xk%=2 and yk%=2

Each graphic output refers first to a 640x400 solution, the parameters of commands like LINE, PLOT, etc. are divided by xk% or yk%

By itself, this method isn't enough to solve all problems with resolution. A few additional points must be considered. For example, the menu bar is not as high in the two color resolutions as in the monochrome resolution. Also, commands like SETCOLOR are forbidden as well.

In low resolution only 40 characters per line are permitted; text outputs must be adjusted accordingly. A special case is the menubar. It should not be more than 40 characters long. That's why menu titles are usually very short.

A menu can never cover more than one fourth of the screen. If it's necessary for the same menu to be used in low resolution and also in medium and high resolution, it can only use a maximum of one eighth of the screen in medium and high resolution.

2.4 Data and Program Structure

In the section about suitable data structure it was pointed out which arrays we need in our program. A point-list exists, and this list consists of three parts, the X-, Y- and Z-coordinates.

2.4.1 Arrays and their Functions

The lines which can be seen in the object drawings have to be kept in an edge table consisting of two arrays. The first array is the starting point, the second array is the end point.

Finally, a plane table is needed, containing the corner points of each plane. These arrays carry the following names in the program:

x(cnt): x-coordinate of point *i*

y(cnt): y-coordinate of point *i*

z(cnt): z-coordinate of point *i*

kb%(cnt): number of point, where i-edge starts.

ke%(cnt): number of point, where i-edge ends

fl%(cnt%,j%): number of point, which is at i-plane the j corner point. In array element fl%(cnt,0) is the number of corner points , which has the i-plane.

Coordinate arrays must be real arrays, because when reducing an object, relevant decimal positions may occur. If no attention is paid to this, a strong reduction with a later enlargement will cause the object to be destroyed.

Edge point numbers and plane corner points can naturally be managed as integers. Now we could use a little trick, in each 4-byte number two numbers are coded. The following example shows how this is done:

A edge is supposed to start at point 29 and end at point 137. The maximum number of manageable numbers is defined by the program as 9999. Both point numbers can be put into one number by using the following piece of code.

```
kb%=29
ke%=137
z%=kb%+10000*ke%
```

Beginning and end points are encoded in the resulting number, 1370029. They can be separated again by using:

```
kb%=Frac(z%/10000)*10000
ke%=Int(z%/10000)
```

As you can see, this trick can save you a lot of storage area, but it happens at the expense of program readability, the length of the program and execution speed. This procedure is only valuable for large arrays when time critical functions are not used.

By the way, this problem can be solved by writing one number into upper two bytes and the other number into the lower bytes of the four-byte integer. This is done by using DPOKE and DPEEK, and the address of variables is found with VARPTR.

This allows for very simple character and calculation operations. Manipulations of the object, such as moving or turning only concern coordinate arrays, but not the edge or plane array. The objects can be constructed by simply connecting the old object list to the new one.

The main trick is that edge and plane arrays can be used as indexes for coordinate arrays. For example, to draw the 20th edge, we can use:

```
LINE   x%(kb%(20)),   y%(kb%(20)),   x%(ke%(20)),
y%(ke%(20))
```

It is absolutely necessary that you understand how this method functions. 20 is the index for the edge array as well as for the list of edge-beginning points and ending points. The developing term *kb%(20)* and *ke%(20)* itself again is an index for other arrays, for the coordinates.

It follows that each object can be drawn as a wire model, by simply writing:

```
FOR cnt%=1 to edges%
 LINE x%(kb%(cnt%)), y%(kb%(cnt%)),
 x%(ke%(cnt%)), y%(ke%(cnt%))
NEXT cnt%
```

In this case the variable *edges%* contains the number of edges in the object.

Similarities occur for the planes, too. The X-coordinate of the third corner point of the 10th plane is:

```
x%(fl%(10,3))
```

Notice that the zero element of the array is used to mark the number of corner points of the plane. Since the array for the plane corner points is a two-dimensional array, the increase of the maximum possible number of corner points eats up quite a bit of storage space.

If the object only contains one plane, with 9 corner points, the field index has to admit the number 9, even if all other planes have only 4 corner points. Unfortunately, this is combined with the waste of some storage area. To avoid such waste, a far more complicated system of object management is necessary.

There are two additional arrays besides these. To be able to combine two objects easily, the two objects must be held in memory. This also simplifies a few other functions. Up until now, all of the arrays are used twice. The first time for the actual object, the second time for the so-called module. This object is the second object in storage, and it exists in the background, because it doesn't appear on screen immediately. Functions like 'put_together' or 'exchange_object_and_module' make it visible.

The array names for the module are *x_mod(), y_mod(), z_mod, kb_mod%(), ke_mod%()* and *fl_mod%()*.

The other existing arrays are not as large and only have small assistance functions. The arrays *x_red()* and *y_red()* pick up the rotation profile from where the rotation elements are created. The arrays each have 50 elements (without the unused zero element).

The arrays *x_trans()* and *y_trans()* have a function in creating the translation elements. Here, only 10 elements are available for each array, which coincides with the maximum number of corner points.

The arrays *plx%()* and *ply%()* are needed for the POLYLINE command, *fl_center%()* contains the center Z-coordinate (distance) for all planes. The three arrays *h1(), h2()* and *h3()* (*h* stands for Helping array) contain the components of normal vectors for all planes. All of these last named arrays are needed for the Hidden-Line Algorithm.

Dimensioning of an object and module arrays are subject to available storage area. Their sizes are marked in the variables *points%*, *edges%*, *planes%* and *max_cornerpoints%*. The names of these variables, hopefully, speak for themselves.

2.4.2 Main Program and PROCEDURE Structure

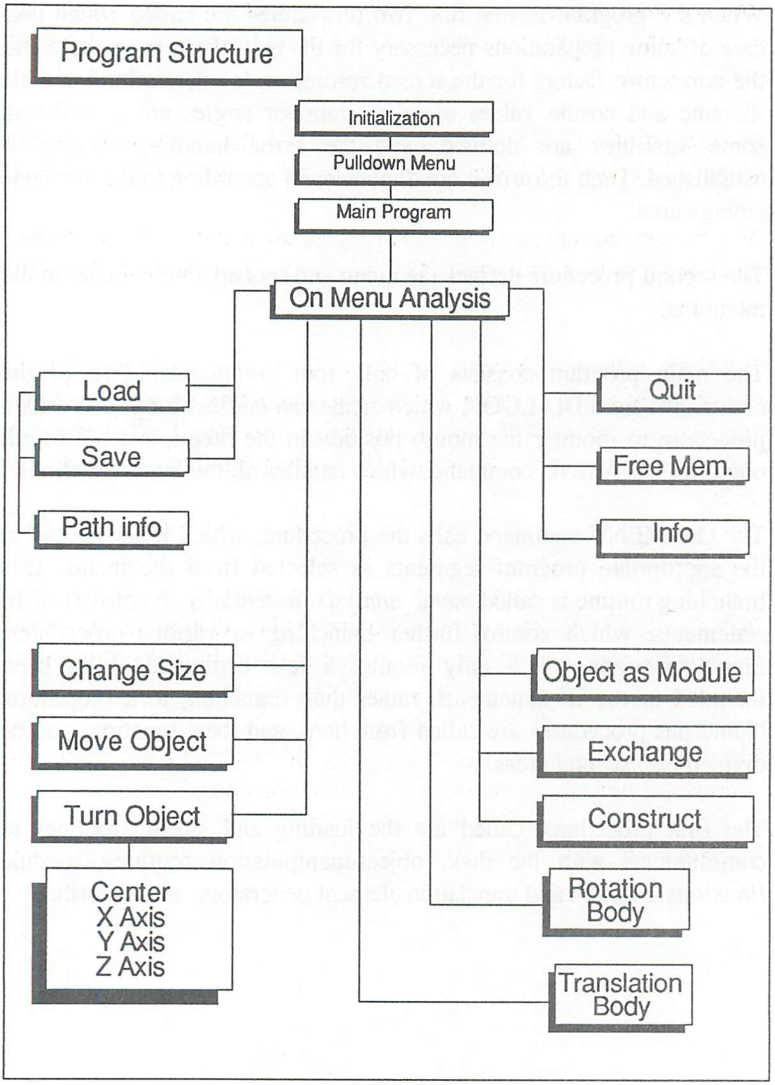
When the program is first run, two procedures are called which take care of some preparations necessary for the rest of the program. First, the correcting factors for the screen resolution are determined, a table for sine and cosine values of whole number angles are established, some variables are defined, and the error handling routine is established. Then the arrays are dimensioned according to the available storage area.

The second procedure defines the menu and sets up a pixel meter in the menu bar.

The main program consists of only four commands. Two of the commands are a DO-LOOP, which creates an endless loop, one calls a procedure to monitor the mouse position in the menu, and the fourth one is an ON MENU command, which handles all the menu functions.

The ON MENU command calls the procedure, which branches out to the appropriate program segments as selected from the menu. This branching routine is called *menu_analysis*. Essentially, it consists of IF statements, which control further branching to various procedures. Some functions which only require a few commands have been included in the IF statements, rather than branching to a procedure. Numerous procedures are called from here, and their functions can be divided into several areas.

The first procedures called are the loading and storing routines to communicate with the disk, object-manipulation routines, module functions, rotation and translation element generators, and so forth.


Program Structure
Initialization
Pulldown Menu
Main Program
On Menu Analysis
Load
Save
Path info
Quit
Free Mem.
Info
Change Size
Move Object
Turn Object
Center
X Axis
Y Axis
Z Axis
Object as Module
Exchange
Construct
Rotation Body
Translation Body

As you can see from this chart, some procedures called up from *menu_analysis*, use additional procedures. Loading and storage routines share a routine called *retain_path* which insures that the last search path of the fileselect box is also adjusted with the next fileselect call.

In some routines, like *draw_object*, we did not write down which routines call them up, because they can be called by many procedures.

As you can see, the structure of this program is quite simple, so new functions can be easily integrated. Just add a new item to the menu which will call the new procedure.

This concludes the theoretical discussion of the program. During development of the program, all thoughts, including set-up of data and program structure, were completed before the first line of program code was written. Now we will begin writing the individual program parts.

2.5 The Main Program

The main program first calls a procedure to place data in the arrays and to initialize some variables. The procedure then initializes the menu structure.

The main loop follows, overseeing the pulldown menu and calling the appropriate procedures, and monitoring the position of the mouse.

```
@preparation
@pulldown_menu
'
DO
 'control menu bar
 ON MENU
 '
 'show mouse coordinates
 @pixelmeter
LOOP
```

2.6 Initializing Section

The initialization section of this program is divided into two procedures, for preparation and managing the pulldown menu. Preparation takes over assigning values to variables, while the menu management routine is exclusively responsible for handling any interaction with the pulldown menu bar.

2.6.1 Arrays and Variables

With the help of XBIOS(4), the actual resolution is determined, and the correction factors defined. It was already explained in a previous section what these factors do. The variable *top%* states, at what Y-pixel the pulldown menu bar ends. Since there could be graphical solutions for which other returned values are reserved from XBIOS(4), the definition of correction factors must be written so that they cannot become zero.

Secondly, a table of sine and cosine values for whole number angles is set up. It speeds up the trigonometric functions, which are used later. (The sine and cosine functions are relatively slow, if you do not have a math coprocessor.) While writing this section, pay attention so index angles can be expressed in degrees.

The table creation can be speeded up by taking advantage of

```
        Sin(W+90)=Cos(W)(W in degrees).
```

Several variables are adjusted after the trigonometric tables are defined; the object representation mode on a wire lattice (mode%=1), the mouse arrow on a small cross and the name of the path for the fileselect boxes. The error handling routine is defined, arrays are dimensioned, and the amount of free memory area has to be determined. When the available storage area is divided by 200, the result is stored in a variable called *count%*. The maximum amount of points and planes is set at *count%*, and the edges are doubled.

Why are twice as many edges allowed than points or planes? Imagine the elements of any object, for instance, a pyramid or a cube. When you count these elements (the number of points, edges and planes) you will find out that the proportion one to two to one makes sense. More puzzling is the question why this ominous variable *count%* is defined as FRE(0)/200. With *count%*, the free storage area is supposed to be divided.

Let us start with the question, how many points can an object have as a maximum, if the empty storage area is known through FRE(0). First, determined how many bytes are needed for a point. All arrays, having anything to do with this object point, must be counted.

Then we need to define a point in X-, Y- and Z-coordinates (3*6 bytes), thus 18 bytes are used. The bytes of edge array are added. Since each point must be paired with two edges, it will consist of two integer edge arrays (2*4=8 bytes), or 26 bytes. Each point is matched with one plane. This plane can have a maximum of 10 corner points, which is, by the way, a relatively arbitrary arrangement. Since each corner point is held in 4 bytes and with each plane containing the number of corner points which need to be saved, the total is 44 bytes.((10+1)*4=44 bytes.) This results in 26+44=70 bytes.

The same memory area is needed again for module arrays, resulting in 140 bytes per object point. Four real number assisting arrays are needed for the hidden-line-algorithm per plane (fl-center, h1, h2, h3), which cover 4*6=24 bytes. Our array is already 140+24 or 164 bytes. Some additional arrays are not dependent on the number of object points, but do need storage space, (the sine and cosine tables and arrays for routine and translation profiles). Moreover the storage area should not be utilized to the last byte, because free space is still needed.

About 200 bytes are needed per object point, as long as we are working without any tricks to save storage area, as was mentioned in the section about data structures. Because of that, the available storage area is divided by 200 and the resulting value is used as maximum number of object points, fixed at the dimensions of the other arrays.

To be able to conduct this adjustment process reasonably, a minimum of storage area has to be available, which can be checked by a IF statement before using the DIM statement.

Now, here is the procedure:

```
PROCEDURE preparation
 '
 ' Determine factors for adjusting resolution
 xk%=1         ! for high resolution
 yk%=1
 high%=19
 IF XBIOS(4)=1    ! medium resolution
  yk%=2
  high%=10
 ENDIF
 IF XBIOS(4)=0    ! low resolution
  xk%=2
  yk%=2
  high%=10
 ENDIF
 '
 ' Table of sine and cosine values
 ' for whole numbered angles
 DIM si(360),co(360)
 arc_to_grad=PI/180
 count%=-1
 REPEAT
  INC count%
  si(count%)=SIN(count%*arc_to_grad)
  co(count%)=COS(count%*arc_to_grad)
 UNTIL count%=360
```

```
'
' Predefine some variables
mode%=1       ! Display Wire Lattice mode
DEFMOUSE 5    ! Mouse as small cross
path$="\*.*"
'
ON ERROR GOSUB error_handler
'
' Conform dimensioning to
' available memory
IF FRE(0)>1800
  '
  ' for rotations and translation elements
  DIM x_rot(50),y_rot(50)
  DIM x_trans(20),y_trans(20)
  '
  ' adjust to memory
  count%=FRE(0)/200
  max_points%=count%
  max_edges%=2*count%
  max_planes%=count%
  max_cornerpoints%=10
  '
  ' for Object
  DIM x(max_points%), y(max_points%)
  DIM z(max_points%)
  DIM kb%(max_edges%),ke%(max_edges%)
  DIM fl%(max_planes%,max_cornerpoints%)
  '
  ' for hatching algorithm
  DIM fl_center(max_planes%)
  DIM plx%(max_cornerpoints%)
  DIM ply%(max_cornerpoints%)
  DIM h1(max_planes%),h2(max_planes%)
  DIM h3(max_planes%)
```

```
    '
    ' For Module
    DIM x_mod(max_points%),y_mod(max_points%)
    DIMz_mod(max_points%)
    DIM kb_mod%(max_edges%),ke_mod%(max_edges%)
    DIM fl_mod%(max_planes%,max_cornerpoints%)
    '
  ELSE
    ALERT 2,"Not enough memory",1," Return ",a%
    EDIT
  ENDIF
RETURN
```

2.6.2 The Pull Down Menu and its Analysis Procedure

Building the array for the pull down menu and installing it is fast. The menu entries are read in a loop, the analysis procedure is determined, and the menu is displayed on the screen.

Two things must be considered. First: If you want to enlarge the number of menu items, you must change the dimensions of the array, *entry$()*. Secondly: No OPENW 0 is used in this procedure. This causes the zero point of display to be in the upper left monitor corner. This makes it possible to include a pixel meter to show the mouse coordinates on the menu bar.

```
PROCEDURE pulldownmenu
  '
  ' make room for menu entries
  ' and read them
  DIM entry$(40)
  count%=-1
  REPEAT
    INC count%
    READ entry$(count%)
  UNTIL entry$(count%)="End of list"
  entry$(count%)=""
```

```
 '
 ' List of pulldown menu entrys
 DATA DESK,Info,--------------------
,1,2,3,4,5,6,""
 DATA File,Load,Save,Memory,Quit,""
 DATA Obj,Mode,Turn,Move,Change
 DATA Resize,Delete,""
 DATA Mod,Object as Module,Exchange
 DATA Put together,""
 DATA Option,Rotation,Translation,Expression,""
 DATA End of list
 '
 ' End of loop
 ON MENU GOSUB menu_analysis
 MENU entry$()
 '
 ' Integrate Pixelmeter
 ' in pulldown menu bar
 IF xk%=1
  PRINT AT(64,1);"X:"   ! x-Pixelmeter
  PRINT AT(72,1);"Y:"   ! y-Pixelmeter
 ENDIF
 '
RETURN
```

The construction of *PROCEDURE menu_analysis* is pretty simple. It consists of the IF THEN ELSE statement which calls the corresponding procedure.

The following procedure is just clerical work.

```
PROCEDURE menu_analysis
 '
 MENU OFF
 m$=entry$(MENU(0))
 '
```

```
IF m$="Info"
 ALERT 1,"3-D|Program",1,"Return ",a%
 DEFMOUSE 5
ENDIF
'
' FILE Menu
IF m$="Load"
 @d_load
ENDIF
'
IF m$="Save"
 @storing
ENDIF
'
IF m$="Memory"
 @memory_area
ENDIF
'
IF m$="Quit"
 @quit
ENDIF
'
' OBJECT Menu
IF m$="Mode"
 ALERT 2,"Representation mode?",1," Wire |
 Solid | Shaded ",mode%
 DEFMOUSE 5
 @object_draw
ENDIF
'
IF m$="Turn"
 @object_rotate
ENDIF
'
IF m$="Move"
 @object_move
ENDIF
'
```

```
IF m$="Resize"
 @object_resize
ENDIF
'
IF m$="Delete"
 ALERT 2,"Delete Object.|Are you sure?",1,"
 Yes | No ",a%
 DEFMOUSE 5
 IF a%=1
  points%=0
  edges%=0
  planes%=0
  @cls
 ENDIF
ENDIF
'
' MODULE Menu
IF m$="Object as Module"
 @object_as_module
ENDIF
'
IF m$="Exchange"
 @exchange
ENDIF
'
IF m$="Put together"
 @put_together
ENDIF
'
' OPTION menu
IF m$="Rotation"
 @rotation_element
ENDIF
'
IF m$="Translation"
 @translation_element
ENDIF
'
```

```
 ' OPTION Menu
 IF m$="Expression"
  @expression
 ENDIF
 '
RETURN
```

2.7 Object Manipulations

This section will explain the routines accessed from the pull down menu. These procedures control turning, moving, change the object size and deleting an object. Since we don't have any routines for creating an object, we will first introduce two replacement procedures for this.

2.7.1 An example Object in Data

In this example, the object created is a cube, whose coordinates and edge- connections are read with a loop from data statements.

```
PROCEDURE example_object
 ' read into point coordinates
 points%=8
 FOR count%=1 TO 8
  READ x(count%),y(count%),z(count%)
 NEXT count%
 ' List of point coordinates
 DATA 100,100,100
 DATA 100,100,200
 DATA 100,200,200
 DATA 100,200,100
 DATA 200,100,100
 DATA 200,100,200
 DATA 200,200,200
 DATA 200,200,100
 '
 ' Read into begin of edge
 ' and end of edge
 edges%=12
 FOR count%=1 TO 12
  READ kb%(count%),ke%(count%)
 NEXT count%
 ' List of edgepoints
 DATA 1,2,2,3,3,4,4,1
 DATA 5,6,6,7,7,8,8,5
 DATA 1,5,2,6,3,7,4,8
RETURN
```

2.7.2 Temporary Drawing Routine

The following drawing routine can be improved for the finished procedure.

It contains some calls, which make a correct representation of the object possible. The procedure *cls* only deletes the screen, without removing the menu bar. Since we did not give an OPENW 0-command, the normal CLS command would clear the screen and unfortunately remove the menu bar. The routines *clipping_on* and *clipping_off* make sure that the object is not drawn in the menu bar. Clipping, in computer graphics, is a function which limits the graphical output to a portion of the screen. The user interface of the Atari ST has a built-in clipping routine, which can be inserted easily into your own programs, thanks to GEM.

```
PROCEDURE draw_object
 @cls
 @clipping_on
 FOR count%=1 TO edges%
  LINE x(kb%(count%)), y(kb%(count%)),
  x(ke%(count%)), y(ke%(count%))
 NEXT count%
 @clipping_off
RETURN
```

The procedure *cls* is simply:

```
PROCEDURE cls
  ' delete screen, but retain menubar
  DEFFILL 1,0
  PBOX 0,top%,639/xk%,399/yk%
RETURN
```

Both procedures to turn the clipping on and off use a VDI-call to store some parameters into the respective arrays. A detailed description can be found in Frank Ostrowski's book, *The GFA BASIC Book*; therefore we will only introduce the listings.

To turn on the clipping:

```
PROCEDURE clipping_on
  '
  ' VDI routine to turn on clipping
  ' which protects the menubar.
  '
  DPOKE CONTRL+2,2
  DPOKE CONTRL+6,1
  '
  DPOKE INTIN,1
  DPOKE PTSIN+0,0
  '
  DPOKE PTSIN+2,high%
  DPOKE PTSIN+4,639/xk%
  DPOKE PTSIN+6,399/yk%
  '
  ' VDI function call
  VDISYS 129
RETURN
```

and to turn of the clipping:

```
PROCEDURE clipping_off
 '
 ' Clipping to the fullest screen size
 ' mouse coordinates can return menubar.
 '
 DPOKE CONTRL+2.2
 DPOKE CONTRL+6,1
 '
 DPOKE INTIN,1
 '
 DPOKE PTSIN+0,0
 DPOKE PTSIN+2,0
 DPOKE PTSIN+4,639/xk%
 DPOKE PTSIN+6,399/yk%
 '
 ' VDI function call
 VDISYS 129
RETURN
```

2.7.3 Rotating the Object

Let's talk about the most difficult routine of object manipulation, which is also the most impressive, the rotation of an object.

The procedure *turn_object* questions the rotation angles around the three volume angles. They are read as strings and then changed into numbers, so the <Return> key can be pressed, to insert a zero.

The routine is constructed so that the rotation angle values have to be between 0 and 360 degrees. You can rewrite them easily in a way, so that even negative declarations are permitted. Secondly, after obtaining the angle a procedure is called to determine the center of the object. These center coordinates are used as the turning point. If the turning angles were correct, procedures are called which carry out the rotation for each angle.

```
PROCEDURE object_rotate
 IF points%>1
  @cls
  PRINT AT(2,5);"Rotating angle around X Axis:
  ";
  INPUT ang$
  rotation_angle_x%=VAL(i$)
  PRINT AT(2,7);"Rotating angle around Y Axis:
  ";
  INPUT ang$
  rotation_angle_y%=VAL(i$)
  PRINT AT(2,9);"Rotating angle around Z Axis:
  ";
  INPUT ang$
  rotation_angle_z%=VAL(i$)
  '
  ' Calculate rotation center.
  @center
  '
  ' If the rotation is within the permitted
  area
  ' rotate it
  IF rotation_angle_x%>0 AND
  rotation_angle_x%<360
   @rotate_x
  ENDIF
  IF rotation_angle_y%>0 AND
  rotation_angle_y%<360
   @rotate_y
  ENDIF
  IF rotation_angle_z%>0 AND
  rotation_angle_z%<360
   @rotate_z
  ENDIF
  '
  ' Redraw object
  @object_draw
  '
 ENDIF
RETURN
```

To calculate the center of the object, the maximum and minimum coordinates along all volume angles have to be determined along with their center values. This procedure is also needed to change the dimensions of an object.

```
PROCEDURE center
  '
  ' Calculate rotation and
  ' change object centers.
  ' In both cases, it's better
  ' to center the object
  '
  ' Starting values
  x_min=x(1)
  y_min=y(1)
  z_min=z(1)
  x_max=x(1)
  y_max=y(1)
  z_max=z(1)
  '
  ' Calculate max and min for
  ' all three dimensions.
  FOR count%=2 TO points%
    x_min=MIN(x(count%),x_min)
    y_min=MIN(y(count%),y_min)
    z_min=MIN(z(count%),z_min)
    x_max=MAX(x(count%),x_max)
    y_max=MAX(y(count%),y_max)
    z_max=MAX(z(count%),z_max)
  NEXT count%
  '
  ' Calculate coordinate centers
  ' along all three dimensions
  center_x=(x_max+x_min)/2
  center_y=(y_max+y_min)/2
  center_z=(z_max+z_min)/2
  '
RETURN
```

The three procedures for rotating the object are constructed in the same way. The formulas used for calculating the rotation have already been introduced.

```
PROCEDURE rotate_x
 '
 ' Cosine and Sine of the angle
 ' (Saves some work later)
 co=co(rotation_angle_x%)
 si=si(rotation_angle_x%)
 '
 ' Turning matrix for rotation of X angle.
 ' The vectors from the object center
 ' to the point coordinates are rotated.
 '
 FOR count%=1 TO points%
  vector_y=y(count%)-center_y
  vector_z=z(count%)-center_z
  y(count%)=co*vector_y-si*vector_z+center_y
  z(count%)=si*vector_y+co*vector_z+center_z
 NEXT count%
 '
RETURN
 '
PROCEDURE rotate_y
 '
 ' Cosine and Sine of the angle
 ' (Saves some work later)
 co=co(rotation_angle_y%)
 si=si(rotation_angle_y%)
 '
 ' Turning matrix for rotation of X angle.
 ' The vectors from the object center to the
 ' point coordinates are rotated.
 '
 FOR count%=1 TO points%
  vector_x=x(count%)-center_x
  vector_z=z(count%)-center_z
  x(count%)=co*vector_x-si*vector_z+center_x
  z(count%)=si*vector_x+co*vector_z+center_z
 NEXT count%
 '
RETURN
'
PROCEDURE rotate_z
 '
 ' Cosine and Sine of the angle
 ' (Saves some work later)
 co=co(rotation_angle_z%)
 si=si(rotation_angle_z%)
 '
 ' Turning matrix for rotation of X angle.
 ' The vectors from the object center to the
 'point coordinates are rotated.
 '
 FOR count%=1 TO points%
  vector_x=x(count%)-center_x
  vector_y=y(count%)-center_y
  x(count%)=co*vector_x-si*vector_y+center_x
  y(count%)=si*vector_x+co*vector_y+center_y
 NEXT count%
 '
RETURN
```

2.7.4 Redimension an Object

For now, the routine to change dimensions, uses three changing factors for axes. Later the center of the object is calculated and the factors are multiplied with the vectors which point from the object center to the points of the object.

```
PROCEDURE object_resize
 @cls
 '
 ' Get dimension change parameter.
 ' By reading as a string, the <Return> key
 ' may be evaluated as a zero.
 ' A number larger than 1 means enlarge the
object.
 ' A number smaller than 1 means reduce the
 ' object.
 ' A minus sign (-) mirrors the object
 '
 PRINT AT(2,5);"Dimension change along X axis:
 ";
 INPUT r$
 x_dimension=VAL(r$)
 PRINT AT(2,7);"Dimension change along Y axis:
 ";
 INPUT r$
 y_dimension=VAL(r$)
 PRINT AT(2,9);"Dimension change along Z axis:
 ";
 INPUT r$
 z_dimension=VAL(r$)
 '
 ' Calculate the center of the Object
 @center
 '
 ' Change dimensions. The vectors between
 ' the center of the object and the point
 ' coordinates are changed.
 '
 ' Dimension change along X Axis
 '
 IF x_dimension<>0
  FOR count%=1 TO points%
   vector_x=x(count%)-center_x
   vector_x=vector_x*x_dimension
   x(count%)=vector_x+center_x
  NEXT count%
 ENDIF
 '
 ' Dimension change along Y Axis
 IF y_dimension<>0
  FOR count%=1 TO points%
   vector_y=y(count%)-center_y
   vector_y=vector_y*y_dimension
   y(count%)=vector_y+center_y
  NEXT count%
 ENDIF
 '
 ' Dimension change along Z Axis
 IF z_dimension<>0
  FOR count%=1 TO points%
   vector_z=z(count%)-center_z
   vector_z=vector_z*z_dimension
   z(count%)=vector_z+center_z
  NEXT count%
 ENDIF
 '
 ' Draw the new object
 @object_draw
 '
RETURN
```

2.7.5 Shifting an Object

The object shifting procedure at first reads the three shifting parameters and simply adds these to the respective point-coordinate.

The zero-point on the monitor must be in the upper left corner, but in our coordinate system, the zero point is in the lower left corner. Therefore the Y-shifting has to be supplied with either a negative sign, or be subtracted, rather than added.

```
PROCEDURE object_move
 IF points%>0
  @cls
  '
  ' Read shift parameter as a string.
  PRINT AT(2,5);"Shift along X axis: ";
  INPUT r$
  shift_x%=VAL(r$)
  PRINT AT(2,7);"Shift along Y axis: ";
  INPUT r$
  shift_y%=VAL(r$)
  PRINT AT(2,9);"Shift along Z axis: ";
  INPUT r$
  shift_z%=VAL(r$)
  '
  ' Move along all axis.
  FOR count%=1 TO points%
   ADD x(count%),shift_x%
   ADD y(count%),-shift_y%
   ADD z(count%),shift_z%
  NEXT count%
  '
  ' Draw new object
  @object_draw
  '
 ENDIF
RETURN
```

2.7.6 Delete Object

The routine to delete an object is naturally very simple. Only the variables, containing the dimensions of the object, have to be set to zero and the screen must be cleared. Since this only concerns a few commands, I did not write my own procedure, but placed these commands in the procedure *menu-analysis*, which was explained earlier.

2.8 The Object Generator and the Drawing Routine

In order to use the object manipulation possibilities, we must add object creation functions into our program.

3D-programs have several methods of creating objects. Most of the time, several of these methods are used in programs. Often programs, provided with module libraries, are complex structures.

Other programs use rudimentary methods of object creation, where each point and line of the object has to be defined individually. An example of this category would be GFA VECTOR.

Users of large computers often have to work with non graphic capable terminals. Large computers are able to create fantastic graphics, but mostly they are set up for many terminals which offer the user limited possibilities, assuming the user does not have a high system clearance. In this case, creating 3D objects consists of the input of number columns.

Our program consists of two object generators available for rotating and translating symmetric elements. Almost all object forms can be created in this manner. Complex objects can be built by putting together these basic forms.

2.8.1 Rotation Elements

Rotation elements are created when a profile is set. These elements are then arranged in several copies around a rotation axis.

Rotations profile:

Resulting rotations element

Rotation axis

The number of copies should be adjustable in order to make this kind of object creation even more flexible. To create hemispheres and similar elements, the option must exist to keep the total of the rotation angles less than 360 degrees.

The maximum number of profile points for rotation should not be larger than 50, or the resulting object may get too large. If the actual object contains less than 50 points, this object can be used as profile.

The procedure *rotation_element* does the following: First it checks whether the actual object can be used. If it can be used, it will ask if the user wants this.

If it is not possible, or the user selects no, the rotation profile must be read by the procedure *read_profile*. The parameters for rotation are read in another procedure. The most complicated routine is the procedure *calculate_rotation_elements*, in which the coordinate lists of the object are created.

Here is listing for the procedure *rotation_element*. It picks up the actual object into the arrays *x_rotate()* and *y_rotate()* or branches into the procedure *read_rotation_profile*.

```
PROCEDURE rotation_element
 '
 ' If the object point number is smaller
 ' than 51 this object may not be used
 ' as the rotation profile.
 IF points%<=50 AND points%>0
  ALERT 2,"Use actual Object?",2," Yes | No
",a%
  DEFMOUSE 5
 ELSE
  a%=2  ! No, do not use
 ENDIF
 '
 ' If the actual object is going to be used
 ' use the coordinates in the arrays, otherwise
 ' otherwise store new data in these arrays.
 IF a%=1
  number_rotate%=points%
  FOR count%=1 TO number_rotate%
   x_rotate(count%)=x(count%)
   y_rotate(count%)=y(count%)
  NEXT count%
 ELSE
  @read_rotation_profile
 ENDIF
 '
 ' Get rotation parameters
 @read_rotation_parameters
 '
 ' Calculate rotation elements
 @calculate_rotation_elements
 '
 ' Redraw object
 @object_draw
 '
RETURN
```

The procedure *read_rotation_profile* fills the arrays *x_rotate()* and *y_rotate()* with profile data. The rotations axis is drawn in the center of the monitor. The points are read in a loop, each point is connected by a line with the preceding point. The loop will end when 50 points are set or the right mouse button is pressed.

The REPEAT UNTIL MOUSEK=0 loop waits for the mouse button to be released. Imagine, the left mouse button has been pressed to indicate that a point has been set. The respective commands in the loop receive the coordinates of the point and then the loop starts over again. If the mouse button would still be pressed, the same point would again be read. Therefore, it is necessary to wait for the release of the mouse button.

```
PROCEDURE read_rotation_profile
 @cls
 LINE 320/xk%,high%,320/xk%,399/yk%
 ' Rotation axis
 count%=0
 REPEAT
  @pixelmeter
  '
  ' Left mouse button pressed means to
  ' set the point, store the coordinates,
  ' and connect with the preceding point.
  ' Pressing the right mouse button, or
  'setting 50 points terminates this procedure.
  ' During this procedure, the mouse cannot be
  ' in the menubar.
  '
  IF MOUSEK=1 AND MOUSEY>20
   INC count%
   x_rotate(count%)=MOUSEX
   y_rotate(count%)=MOUSEY
   IF count%>1
    LINE x_rotate(count%-1), y_rotate(count%-
    1), x_rotate(count%), y_rotate(count%)
   ELSE
    PLOT x_rotate(count%),y_rotate(count%)
   ENDIF
```

```
    '
    ' Wait for release of mouse button
    REPEAT
    UNTIL MOUSEK=0
   ENDIF
  UNTIL MOUSEK=2 OR count%=50
  '
  number_rotate%=count%
  '
  FOR count%=1 TO number_rotate%
   MUL x_rotate(count%),xk%
   MUL y_rotate(count%),yk%
  NEXT count%
RETURN
```

Reading the rotation parameters is done with two input instructions. The only check is whether the parameters are within the legal number area. The condition:

FRAC(total_angle%/number_planes%)=0

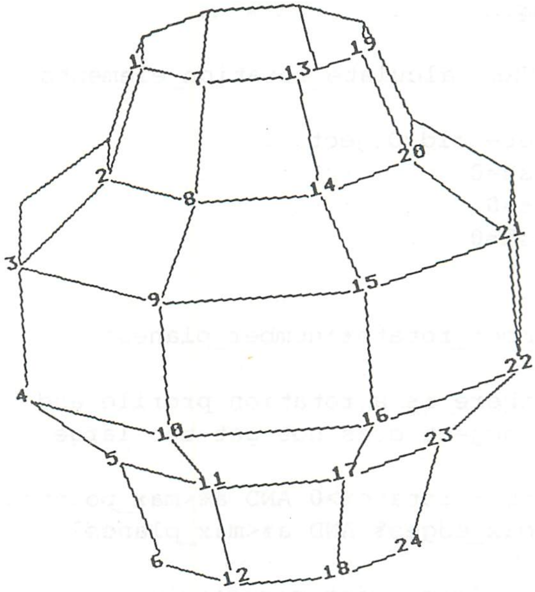
is necessary, so the rotation angles are whole numbers. In our sine-cosine tables only values for whole numbers angles are included.

```
PROCEDURE read_rotation_parameters
  @cls
  PRINT AT(2,4);"Adjust parameters for
  rotation:"
  '
  ' Get rotation angles
  REPEAT
   PRINT AT(2,7);"Rotation Angle
   (90,180,270,360): ";
   INPUT r$
   total_angle%=VAL(r$)
  UNTIL total_angle%>0 AND total_angle%<=360 AND
FRAC(total_angle%/90)=0
  '
  ' Get number of rotation angles
```

```
 REPEAT
  PRINT AT(2,10);"Total rotation angle must be"
  PRINT AT(2,11);"evenly divisible by rotation
  planes."
  PRINT AT(2,12);"Number of rotation planes: ";
  INPUT r$
  number_planes%=VAL(r$)
  number_planes%=MAX(number_planes%,2)
  IF total_angle%=360
   through%=number_planes%
  ELSE
   through%=number_planes%-1
  ENDIF
 UNTIL number_planes%>2 AND
 FRAC(total_angle%/through%)=0
 '
RETURN
```

The following procedure calculates the finished element from the profile and parameters. You must determine the points, edges and planes list.

The following figure shows the consecutive numbering in the finished element. The points in a rotation plane must be counted from top to bottom, until the lowest one is reached, and then continued in the next turning plane.



In the following procedure, the old object is deleted by adjusting the dimension specifications to zero. Then, if the resulting object is not too large, the object is redrawn.

The points of the rotation profile are rotated in a loop around the Y-axis, until the total rotation angle is reached. With each part rotation, the actual position of the rotation profile is adapted into the pointlist.

Next, the edge list is created. The lines between the rotation planes must be determined and then the lines within the rotation planes. Notice in the preceding illustration the connections between the points.

In case the total angle was 360 the object must be closed by connecting the first with the last rotation plane.

Finally, the plane list is completed. Each plane has four corner points. Again, refer to the illustration to understand the method of corner point sequencing.

```
PROCEDURE calculate_rotation_elements
 '
 ' Delete old Object
 points%=0
 edges%=0
 planes%=0
 '
 '
 a%=number_rotate%*number_planes%
 '
 ' If there is a rotation profile and
 ' the object does not get too large
 '
 IF number_rotate%>0 AND a%<max_points% AND
 2*a%<max_edges% AND a%<max_planes%
  '
  ' Calculate point coordinates
  IF total_angle%=360
   angle_interval%=total_angle%/number_planes%
   til%=total_angle%-angle_interval%
  ELSE
   angle_interval%=total_angle%/(number_planes%
   -1)
   til%=total_angle%
  ENDIF
  FOR angle%=0 TO til% STEP angle_interval%
   FOR count%=1 TO number_rotate%
    INC points%
    ' Vector to be rotated; 320 = rotation axis
    x_rotate=x_rotate(count%)-320
    x(points%)=co(angle%)*x_rotate+320
    y(points%)=y_rotate(count%)
    z(points%)=si(angle%)*x_rotate
   NEXT count%
  NEXT angle%
  '
```

```
' Determine lines with rotation plane
IF number_rotate%>1
 FOR flat%=0 TO number_planes%-1
  FOR count%=1 TO number_rotate%-1
   INC edges%
   kb%(edges%)=count%+flat%*number_rotate%
   ke%(edges%)=kb%(edges%)+1
  NEXT count%
 NEXT flat%
ENDIF
'
' Determine lines between rotation planes
FOR flat%=0 TO number_planes%-2
 FOR count%=1 TO number_rotate%
  INC edges%
  kb%(edges%)=count%+flat%*number_rotate%
  ke%(edges%)=kb%(edges%)+number_rotate%
 NEXT count%
NEXT flat%
'
' If total angle is 360, the first
' rotation plane must be connected
' to the last one.
IF total_angle%=360
 FOR count%=1 TO number_rotate%
  INC edges%
  kb%(edges%)=count%+(number_planes%-
  1)*number_rotate%
  ke%(edges%)=count%
 NEXT count%
ENDIF
```

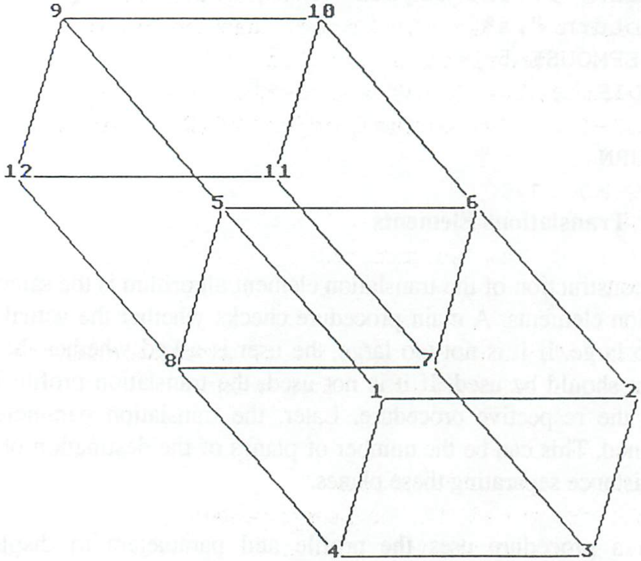
```
 '
 ' Determine planes
 IF number_rotate%>1
  FOR flat%=0 TO number_planes%-2
   FOR count%=1 TO number_rotate%-1
    INC planes%
    fl%(planes%,0)=4
    fl%(planes%,1)=count%+number_rotate%*flat%
    fl%(planes%,2)=count%+number_rotate%*flat%
    +1
    fl%(planes%,3)=count%+number_rotate%*(flat
    %+1)+1
    fl%(planes%,4)=count%+number_rotate%*(flat
    %+1)
   NEXT count%
  NEXT flat%
  '
  ' If total angle is 360, planes must be
  ' inserted between the first and last
  ' rotation planes.
  IF total_angle%=360
   FOR count%=1 TO number_rotate%-1
    INC planes%
    fl%(planes%,0)=4
    fl%(planes%,1)=count%+(number_planes%-
    1)*number_rotate%
    fl%(planes%,2)=count%+(number_planes%-
    1)*number_rotate%+1
    fl%(planes%,3)=count%+1
    fl%(planes%,4)=count%
   NEXT count%
  ENDIF
  '
 ENDIF
ELSE
```

```
   '
   ' Inform user if object is too large.
   ALERT 1,"The Object would be too large.",1,"
   Return ",a%
   DEFMOUSE 5
  ENDIF
  '
RETURN
```

2.8.2 Translations Elements

The construction of the translation element algorithm is the same as for rotation elements. A main procedure checks whether the actual object is too large. If it is not too large, the user is asked whether the actual object should be used. If it is not used, the translation profile is read from the respective procedure. Later, the translation parameters are obtained. This can be the number of planes of the destination object or the distance separating these planes.

Then a procedure uses the profile and parameters to display the finished translation element. The following illustration shows the sequence for numbering the points. This should help you understand the procedure *calculate_translation_element.*



These routines are the similar to the ones described in the last chapter.

```
PROCEDURE translation_element
 '
 ' If the number of object points is
 ' less than 11 the object may be used
 ' as a translation profile.
 IF points%<=10 AND points%>0
  ALERT 2,"Use actual object?",2," Yes | No
  ",a%
  DEFMOUSE 5
 ELSE
  a%=2  ! No, do not use
 ENDIF
 '
 ' If actual object is used, use coordinates
 ' in the arrays for generation.
 '
 IF a%=1
  number_trans%=points%
  FOR count%=1 TO number_trans%
   x_trans(count%)=x(count%)
   y_trans(count%)=y(count%)
  NEXT count%
 ELSE
  @read_translation_profile
 ENDIF
 '
 ' Read parameters for object translation
 @read_translation_parameter
 '
 ' Calculate translation object
 @calculate_translation_element
 '
 ' Draw new object
 @object_draw
 '
RETURN
'
```

```
PROCEDURE read_translation_profile
 @cls
 count%=0
 REPEAT
  @pixelmeter
  '
  ' Press left mouse button to set point,
  ' store coordinates, and connect with
  ' previous point.
  ' Pressing the right mouse button, or
  ' setting 10 points
  ' means the profile is finished
  ' During this procedure the mouse cannot be
  ' in the menu bar.
  '
  IF MOUSEK=1 AND MOUSEY>20
   INC count%
   x_trans(count%)=MOUSEX
   y_trans(count%)=MOUSEY
   IF count%>1
    LINE x_trans(count%-1),y_trans(count%-
    1),x_trans(count%),y_trans(count%)
   ELSE
    PLOT x_trans(count%),y_trans(count%)
   ENDIF
   '
   ' Wait until the mouse button is released
   REPEAT
   UNTIL MOUSEK=0
  ENDIF
 UNTIL MOUSEK=2 OR count%=10
 '
 number_trans%=count%
 '
 FOR count%=1 TO number_trans%
  MUL x_trans(count%),xk%
  MUL y_trans(count%),yk%
 NEXT count%
 '
RETURN
'
PROCEDURE read_translation_parameter
 @cls
 PRINT AT(2,4);"Adjust translation parameters:"
 '
 ' Determine the distance of the planes
 REPEAT
  PRINT AT(2,7);"Distance of the planes: ";
  INPUT r$
  plane_distance%=VAL(r$)/xk%
 UNTIL plane_distance%>0 AND
FRAC(plane_distance%)=0
 '
 ' Determine the number of planes
 REPEAT
  PRINT AT(2,9);"Number of translation planes:
  ";
  INPUT r$
  number_transplanes=VAL(r$)
 UNTIL number_transplanes>0 AND
FRAC(number_transplanes)=0
 '
RETURN
'
```

```
PROCEDURE calculate_translation_element
 '
 ' Delete old object
 points%=0
 edges%=0
 planes%=0
 '
 a%=number_trans%*number_transplanes
 '
 ' If profile exists, and object is
 ' not too large
 IF number_trans%>0 AND a%<max_points% OR
2*a%<max_edges% OR a%+2<max_planes%
  '
  ' Calculate point coordinates
  distance%=0
  FOR flat%=1 TO number_transplanes
   FOR count%=1 TO number_trans%
    INC points%
    x(points%)=x_trans(count%)
    y(points%)=y_trans(count%)
    z(points%)=distance%
   NEXT count%
   ADD distance%,plane_distance%
  NEXT flat%
  '
  ' Edges within translation planes
  IF number_trans%>1
   FOR flat%=0 TO number_transplanes-1
    FOR count%=1 TO number_trans%-1
     INC edges%
     kb%(edges%)=count%+flat%*number_trans%
     ke%(edges%)=count%+flat%*number_trans%+1
    NEXT count%
   NEXT flat%
  ENDIF
```

```
'
' If more than two points are in one plane
' the first and last point must be connected.
'
IF number_trans%>2
 FOR flat%=0 TO number_transplanes-1
  INC edges%
  kb%(edges%)=number_trans%*flat%+1
  ke%(edges%)=number_trans%*flat%+
  number_trans%
 NEXT flat%
ENDIF
'
' Determine edges between translation planes
'
IF number_transplanes>1
 FOR flat%=1 TO number_transplanes-1
  FOR count%=1 TO number_trans%
   INC edges%
   kb%(edges%)=count%+number_trans%*(flat%-1)
   ke%(edges%)=count%+number_trans%*flat%
  NEXT count%
 NEXT flat%
ENDIF
'
' Determine planes within the transplanes.
' This is important for the
' cross-hatching algorithm.
'
IF number_trans%>2
 FOR flat%=0 TO number_transplanes-1
  INC planes%
  fl%(planes%,0)=number_trans%
  FOR count%=number_trans% DOWNTO 1
   fl%(planes%,count%)=count%+(flat%*number_t
   rans%)
  NEXT count%
 NEXT flat%
ENDIF
```

```
'
' Determine planes between transplanes.
' This is important for the
' cross-hatching algorithm.
'
IF number_trans%>1 AND number_transplanes>1
 FOR flat%=0 TO number_transplanes-2
  FOR count%=1 TO number_trans%-1
   INC planes%
   fl%(planes%,0)=4
   fl%(planes%,1)=flat%*number_trans%+count%
   fl%(planes%,2)=flat%*number_trans%+count%+
   1
   fl%(planes%,3)=(flat%+1)*number_trans%+cou
   nt%+1
   fl%(planes%,4)=(flat%+1)*number_trans%+cou
   nt%
  NEXT count%
 NEXT flat%
ENDIF
'
' Insert planes between the first and last
' points of neighboring planes. The
' sequence of the points
' is significant for the
' cross-hatching algorithm.
'
IF number_trans%>2 AND number_transplanes>1
 FOR flat%=0 TO number_transplanes-2
  INC planes%
  fl%(planes%,0)=4
  fl%(planes%,1)=(flat%+1)*number_trans%+1
  fl%(planes%,2)=(flat%+1)*number_trans%+numb
  er_trans%
  fl%(planes%,3)=flat%*number_trans%+number_t
  rans%
  fl%(planes%,4)=flat%*number_trans%+1
 NEXT flat%
ENDIF
'
```

```
 ELSE
 ' Inform user if object would become
 ' too large.
  ALERT 1,"The object would be too large.",1,"
  Return ",a%
  DEFMOUSE 5
 ENDIF
 '
RETURN
```

2.8.3 The Drawing Routine

Now that we are able to create and manipulate very impressive objects, we should improve the representational possibilities and complete the final drawing routine. It should not only display the object as a wire model, but also hatching should be possible as well as diagraming the covered edges.

Altogether there will be three display modes available. The first one is a simple wire lattice mode; the second one is a mode where the edges are covered with planes. In a third mode it is possible to represent the objects with shading. All of these modes should adjust to the actual screen resolution.

The drawing routine is as follows: First the screen is deleted and clipping turned on, then the wire lattice model is drawn. A factor of +0.5 is used with the LINE and POLYFILL commands to round the coordinates. Because the line command only uses the non-decimal positions; the statement:

```
Line 5.9,10,5.9,20
```

would be drawn on the X-coordinate of 5. The variable *mode%* contains information about which mode is to be used. The possible values are:

mode%=1 : Only wire model

mode%=2 : Hidden-Line

mode%=3 : Hidden-Line with hatching

The primitive Hidden-Line Algorithm (Hidden-Surface) begins, when *mode%* is larger than one and planes exist. It calculates the average distance (Z-coordinate) for each plane and sorts them by descending distances with a normal quicksort routine.

The masking of covered planes is not perfect. With the selection of another sorting criterion, i.e. minimum or maximum Z-coordinates instead of the average one, Hidden-Line Errors can be obtained.

If shading (hatching) is not supposed to take place, *mode%* is three, the planes are displayed by POLYFILL. If the figure should be hatched, a vector, vertical to the plane, must be calculated for each plane. The angle between this vector and the light vector will later determine the brightness of the plane. The principle of this procedure has already been explained.

The vector which is vertical to the plane is stored in the variables *H1()*, *H2()* and *H3()*. The source of light is adjusted on (1,1,1). The output of planes is also done with POLYFILL.

```
PROCEDURE object_draw
 '
 ' Draws object completely depending on the
 ' drawing mode selected.
 '
 @cls
 '
 ' Turn on clipping to protect menu bar
 @clipping_on
 '
 ' Draw wire lattice model of object.
 FOR count%=1 TO edges%
   x1%=x(kb%(count%))/xk%+0.5
   LINE x1%, y(kb%(count%))/yk%+
0.5,x(ke%(count%)) /xk%+0.5,
y(ke%(count%))/yk%+0.5
 NEXT count%
```

```
'
' If a primitive Hidden-line should follow
' and planes exist calculate average distance
' (Z-coordinate) of the planes.
'
IF mode%>1 AND planes%>0
 '
 ARRAYFILL fl_center(),0
 FOR count%=1 TO planes%
  FOR j%=1 TO fl%(count%,0)
   ADD fl_center(count%),z(fl%(count%,j%))
  NEXT j%
  DIV fl_center(count%),fl%(count%,0)
 NEXT count%
 '
 ' Sort planes by distance (descending order)
 GOSUB quicksort(1,planes%)
 '
 ' In case of shading (hatching), calculate
 ' normal vectors for planes
 '
 IF mode%=3
  FOR count%=1 TO planes%
   '
   x1=x(fl%(count%,1))
   y1=y(fl%(count%,1))
   z1=z(fl%(count%,1))
   x2=x(fl%(count%,2))
   y2=y(fl%(count%,2))
   z2=z(fl%(count%,2))
   x3=x(fl%(count%,3))
   y3=y(fl%(count%,3))
   z3=z(fl%(count%,3))
   '
   ' Coordinates of normal planes.
   h1(count%)=(y2-y1)*(z3-z1)-(z2-z1)*(y3-y1)
   h2(count%)=(z2-z1)*(x3-x1)-(x2-x1)*(z3-z1)
   h3(count%)=(x2-x1)*(y3-y1)-(y2-y1)*(x3-x1)
  NEXT count%
 ELSE
 '
 ' Fill pattern if hatching is not used.
 DEFFILL 1,0
ENDIF
'
FOR count%=1 TO planes%
 '
 ' Create arrays for POLYFILL
 FOR j%=1 TO fl%(count%,0)
  plx%(j%-1)=x(fl%(count%,j%))/xk%+0.5
  ply%(j%-1)=y(fl%(count%,j%))/yk%+0.5
 NEXT j%
 '
 ' Close POLYFILL arrays
 plx%(fl%(count%,0))=x(fl%(count%,1))/xk%+0.5
 ply%(fl%(count%,0))=y(fl%(count%,1))/yk%+0.5
 '
 ' Fill pattern if hatching is used
 IF mode%=3
  '
  ' Determine length of normal plane vector
  h1=h1(count%)^2+h2(count%)^2+h3(count%)^2
  IF h1>0
   plane_vector_length=SQR(h1)
  ELSE
   plane_vector_length=1
  ENDIF
  '
  ' Cosine of the angle between normal plane
  ' vector and light rays. (Light is
  ' projected from the left top front.
  co_angle=(h1(count%)+h2(count%)+h3(count%))
  /plane_vector_length
  '
  ' Calculate fill pattern from the cosine of
  ' this angle.
  colour%=(co_angle+1)*3+3
```

```
      '
      ' Fill pattern parameter boundaries
      IF colour%<1
       colour%=1
      ENDIF
      IF colour%>6
       colour%=6
      ENDIF
      '
      ' Adjust fill pattern
      DEFFILL 1,2,colour%
    ENDIF
    '
    ' Display plane
    POLYFILL fl%(count%,0),plx%(),ply%()
    '
  NEXT count%
  '
 ENDIF     ! If mode%>1 and planes%>0
 '
 @clipping_off
RETURN
```

GFA BASIC Version 3.0 users can take advantage of the QSORT command, but for other Versions of GFA BASIC, here is an example of the quicksort algorithm.

```
PROCEDURE quicksort(l_index%,r_index%)
 '
 ' Sort planes. Planes with larger than average
 ' distance go to the 'top' of the plane
 ' list. This is a non-optimized recursive
 ' Quicksort.
 '
 count%=l_index%
 j%=r_index%
 border=fl_center(INT((l_index%+r_index%)/2))
 REPEAT
  WHILE fl_center(count%)>border
   INC count%
  WEND
  WHILE border>fl_center(j%)
   DEC j%
  WEND
  IF count%<=j%
   '
   ' Exchange planes
   til%=MAX(fl%(count%,0),fl%(j%,0))
   FOR k%=0 TO til%
    SWAP fl%(count%,k%),fl%(j%,k%)
   NEXT k%
   SWAP fl_center(count%),fl_center(j%)
   '
   INC count%
   DEC j%
  ENDIF
 UNTIL count%>j%
 IF l_index%<j%
  GOSUB quicksort(l_index%,j%)
 ENDIF
 IF count%<r_index%
  GOSUB quicksort(count%,r_index%)
 ENDIF
RETURN
```

2.9 Module Operations

The module storage consists of the arrays *x_mod(), y_mod(), z_mod(), kb_mod(), ke_mod()* and *fl_mod()*. A copy of the object is stored in these arrays. The existence of such a second object storage area allows several practical functions.

Our program will use the three primary functions. The procedure *object_as_module* makes it possible to put a copy of the object into the module storage area. This is practical if you want to take a look at the actual object from different perspectives, and if you want to restore the object to its original condition. The procedure *object_as_module* is used for this purpose, while the procedure *exchange* is used to restore the object.

The procedure *exchange* swaps the contents of object and the module storage area. Another possible application is even more important. If you were to use the program constantly to create room furnishings, for instance, soon you will have created a small library of tables, chairs, lamps and other furniture. A room could be furnished simply by setting the prefabricated furniture pieces into the room.

Of course, it may happen that a piece of furniture is too large. the procedure would then be as follows: the room is buffered by the procedure *object_as_module*. Then the piece of furniture is loaded and converted to the correct size. Now select the procedure *exchange* and the item of furniture may be placed into the room.

This insertion is done with the most important module function, the assembling. The module can be put into the object with the mouse. How that is done is explained in the listing that will follow.

2.9.1 Object as Module

This function copies all object arrays into the module arrays. Dimension details are copied into the respective variables of the module and then the point coordinates, edge, and plane arrays are copied successively. For array copying, the fast method with BMOVE is used. You will find more information about this copying method in the first chapter.

```
PROCEDURE object_as_module
'
' This procedure places a copy of the object
' into the module storage area.
'
' Object size in Module storage area
 mod_points%=points%
 mod_edges%=edges%
 mod_planes%=planes%
'
' Copy point coordinate array.
'
 IF points%>0
  BMOVE LPEEK(ARRPTR(x()))+10,
  LPEEK(ARRPTR(x_mod()))+10, points%*6
  BMOVE LPEEK(ARRPTR(y()))+10,
  LPEEK(ARRPTR(y_mod())+10, points%*6
  BMOVE LPEEK(ARRPTR(z()))+10,
  LPEEK(ARRPTR(z_mod()))+10, points%*6
 ENDIF
'
' Copy edge array
'
 IF edges%>0
  BMOVE LPEEK(ARRPTR(kb%()))+8,
  LPEEK(ARRPTR(kb_mod%()))+8, edges%*4
  BMOVE LPEEK(ARRPTR(ke%()))+8,
  LPEEK(ARRPTR(ke_mod%()))+8, edges%*4
 ENDIF
 '
 ' Copy plane array
 '
 IF planes%>0
  number%=max_planes%*(max_cornerpoints%+1)*4
  BMOVE LPEEK(ARRPTR(fl%()))+12,
  LPEEK(ARRPTR(fl_mod%()))+12, number%
 ENDIF
 '
RETURN
```

2.9.2 Exchange of Object and Module

The exchange of arrays in GFA BASIC is very simple, since the SWAP command can also be used with arrays. The following routine is therefore so simple that it does not require any further explanation.

```
PROCEDURE exchange
 '
 ' Exchange the contents of the object and
 ' module storage area
 '
 ' Exchange point arrays and their dimensions
 '
 SWAP points%,mod_points%
 SWAP x(),x_mod()
 SWAP y(),y_mod()
 SWAP z(),z_mod()
 '
 ' Exchange edge arrays and their dimensions
 '
 SWAP edges%,mod_edges%
 SWAP kb%(),kb_mod%()
 SWAP ke%(),ke_mod%()
 '
 ' Exchange plane arrays and their dimensions
 '
 SWAP planes%,mod_planes%
 SWAP fl%(),fl_mod%()
 '
 ' Draw former module
 @object_draw
 '
RETURN
```

2.9.3 Composition of Object and Module

The composition is, from my viewpoint, the most important function of the program. This is important, because it allows the creation of extremely complex objects.

Basically this function does not have any more to do than placing the arrays on the module point coordinates and module edges and planes onto the respective object, as long as the respective object does not get too large.

You might think that this suspension can be done with BMOVE and would therefore be very fast. Unfortunately, this is not completely correct. This goes for the point coordinate lists, where it would be enough to put the storage area of module coordinates in the place occupied by the object coordinates arrays, so that the module coordinates are directly behind the object coordinates.

This is not possible with the edges and plane arrays because the number of points of the module change when they are placed on the object. One must be sure that the list of points of the module is attached to the object point list and the value of each module point must be raised by the number of points already available in the object. This number carries the name *old_point_numbers%* in the program.

From this you can conclude that edge and plane lists can be copied within a loop. To keep the listing uniform, I have also copied the module coordinate arrays within a loop. So much for the second half of the listing.

The user must be able to select which object the module should be set to. The dimension of the module should be shown with a box. To be able to position it along all three volume coordinates, an additional trick has to be used.

Within the normal view from the front, the user can state only from which X and on which Y coordinates the object should be positioned, it is not possible to say on which Z coordinate, or even better, how far the object should be away from monitor surface.

The following solution is used: the user marks the X and Y coordinates from the front view. Then the object is rotated 90 degrees and two lines show the Z dimensions of the module. With the help of these lines the user is also able to give the distance of the Z coordinates of the module.

The drawing of the movable box and the pair of lines to give the size of the module is done with two subroutines, *move_box* and *move_lines*. They return the mouse coordinates in the variables *mx%*, *my%*, and *mz%*. The variable *mz%* contains the X coordinate of the mouse, from which the Z coordinate of the module position can be determined.

You need to be able to show the dimension of the module. This is determined in the procedure *module_expansions*. To set the module in the correct position, the variables *x_shift*, *y_shift* and *z_shift* are calculated from the old module coordinates and from the user selected target position, and are then subtracted from the old module coordinates.

```
PROCEDURE assembly
  '
  ' This procedure permits the Object and Module
  ' to be assembled, if a Module exists.
  '
  IF mod_points%>1
    '
    ' Copies values of long variable names
    ' into short variable names.
    ' This is only being done to simplify
    ' listing the IF conditions.
    '
    mp%=mod_points%
    mk%=mod_edges%
    mf%=mod_planes%
    maf%=max_planes%
    '
```

```
' If the resulting object is not too large,
' assembling is permitted, otherwise an
' Alert Box is displayed.
'
IF points%+mp%>max_points% OR
edges%+mk%>max_edges% OR planes%+mf%>maf%
 ALERT 1,"Not enough Memory!",1," Return ",a%
ELSE
 '
 ' Determine size of the module
 '
 @module_dimensions
 '
 ' Establish X and Y Position of the Module
 @move_box
 '
 ' Use the selected position to do correction
 '
 x_shift=x_mod_min-mx%*xk%
 y_shift=y_mod_min-my%*yk%
 FOR count%=1 TO mod_points%
  SUB x_mod(count%),x_shift
  SUB y_mod(count%),y_shift
 NEXT count%
 '
 ' Draw object from Side view
 @cls
 FOR count%=1 TO edges%
  '
  ' Center the object on the screen
  '
  za=(z(kb%(count%))+320)/xk%
  ze=(z(ke%(count%))+320)/xk%
  LINE za,y(kb%(count%))/yk%, ze,
  y(ke%(count%))/yk%
 NEXT count%
 '
 ' Determine Z Position of the Module
 '
 @move_lines
```

```
'
' Use selected position to find Z coordinate
'
z_shift=z_mod_min-mz%*xk%+320
FOR count%=1 TO mod_points%
 SUB z_mod(count%),z_shift
NEXT count%
'
' Append Module points to Object point list
'
old_point_numbers%=points%
FOR count%=1 TO mod_points%
 INC points%
 x(points%)=x_mod(count%)
 y(points%)=y_mod(count%)
 z(points%)=z_mod(count%)
NEXT count%
'
' Append Module edges to Object edge list
'
FOR count%=1 TO mod_edges%
 INC edges%
 kb%(edges%)=kb_mod%(count%)+
 old_point_numbers%
 ke%(edges%)=ke_mod%(count%)+
 old_point_numbers%
NEXT count%
'
' Append Module planes to Object planes list
'
FOR count%=1 TO mod_planes%
 INC planes%
 fl%(planes%,0)=fl_mod%(count%,0)
 FOR j%=1 TO fl_mod%(count%,0)
  fl%(planes%,j%)=fl_mod%(count%, j%)+
  old_point_numbers%
 NEXT j%
NEXT count%
```

```
    '
    ' Draw newly assembled Object
    '
    @object_draw
    '
   ENDIF
  ELSE
   ' In case Mod_points%=0
   ALERT 1,"No Module available.",1," Return
   ",a%
  ENDIF
RETURN
```

The three assisting procedures used are relatively simple.

The procedure *module_expand* calculates the minimum and maximum of the three coordinate arrays and determines from there the dimension of the module.

```
PROCEDURE module_dimensions
 '
 ' Start work
 x_mod_min=x_mod(1)
 y_mod_min=y_mod(1)
 z_mod_min=z_mod(1)
 x_mod_max=x_mod(1)
 y_mod_max=y_mod(1)
 z_mod_max=z_mod(1)
 '
 ' Determine Maximum and Minimum dimensions.
 '
 FOR count%=2 TO mod_points%
  x_mod_min=MIN(x_mod(count%),x_mod_min)
  y_mod_min=MIN(y_mod(count%),y_mod_min)
  z_mod_min=MIN(z_mod(count%),z_mod_min)
  x_mod_max=MAX(x_mod(count%),x_mod_max)
  y_mod_max=MAX(y_mod(count%),y_mod_max)
  z_mod_max=MAX(z_mod(count%),z_mod_max)
 NEXT count%
```

```
  '
  ' Determine expansion from Max and Min.
  '
  x_expansion=x_mod_max-x_mod_min
  y_expansion=y_mod_max-y_mod_min
  z_expansion=z_mod_max-z_mod_min
  '
RETURN
```

The next procedure draws a rectangle the size of X and Y expansion of the module. This rectangle can be moved with the mouse. It can be removed by pressing the mouse button.

One problem with this procedure is that the box to be moved is not drawn over the menubar, but disappears under it instead. Even then, the coordinates displayed by the pixelmeter should be possible. The clipping is constantly turned on and off during movements of the box. As you can see, this is rather fast.

```
PROCEDURE move_box
  '
  ' This procedure draws a box with the  X and Y
  ' dimensions of the module. The box may be
  ' moved by using the mouse.
  '
  ' Define line and Graphics mode
  DEFLINE 6
  GRAPHMODE 3
  '
  ' Starting values outside of clipping
  ' rectangle invisible during first pass.
  '
  old_mx%=640
  old_my%=400
```

```
 '
 ' Box may be moved until mouse button is
 ' pressed
 '
 REPEAT
  mx%=MOUSEX
  my%=MOUSEY
  '
  ' Draw only if mouse has been moved.
  ' (Eliminates flicker.)
  IF mx%<>old_mx% OR my%<>old_my%
   @clipping_on
   BOX old_mx%, old_my%, old_mx% +
   x_expansion/xk%, old_my% + y_expansion/yk%
   BOX mx%, my%,mx% + x_expansion/xk%,my% +
   y_expansion/yk%
   old_mx%=mx%
   old_my%=my%
   @clipping_off
  ENDIF
  '
  ' Show mouse coordinates
  @pixelmeter
 UNTIL MOUSEK
 '
 ' Redefine line mode and graphics type
 GRAPHMODE 1
 DEFLINE 1
 '
 ' Wait for mouse button release
 REPEAT
 UNTIL MOUSEK=0
RETURN
```

The procedure *move_lines* does something similar to *move_box*. Two lines are removed which have the size of the Z dimensions of the module. The removal point, after pressing the mouse button is in the variable *mz%*.

```
PROCEDURE move_lines
 '
 ' Draws a pair of lines the size of the Z
 ' dimension, which can be moved by
 ' the mouse.
 '
 ' Define line type and graphics mode
 DEFLINE 6
 GRAPHMODE 3
 '
 ' Starting value outside of clipping rectangle
 ' therefore, not shown during first pass.
 '
 old_mz%=640
 '
 REPEAT
  mz%=MOUSEX
  '
  ' Only draw new object if mouse
  ' has been moved.
  '
  IF mz%<>old_mz%
   @clipping_on
   LINE old_mz%,high%,old_mz%,399/yk%
   LINE old_mz%+z_expansion/xk%, high%,
   old_mz%+z_expansion/xk%, 399/yk%
   LINE mz%,high%,mz%,399/yk%
   LINE mz%+z_expansion/xk%, high%,
   mz%+z_expansion/xk%, 399/yk%
   old_mz%=mz%
   @clipping_off
  ENDIF
  '
  ' Show mouse coordinates
  @pixelmeter
 UNTIL MOUSEK
 '
```

```
  ' Define line type and graphics mode
  '
  GRAPHMODE 1
  DEFLINE 1
  '
  ' Wait until mouse button is released
  REPEAT
  UNTIL MOUSEK=0
RETURN
```

2.10 Saving and Loading Objects

Loading and saving objects consists only of writing the dimension details and the respective arrays to the disk, or reading them from the disk. Naturally, this should be as rapid as possible. We already know that there are different ways to optimize the procedures depending on the size of the numbers and the storage media.

In the case of whole numbers with a large absolute amount, or large real number arrays with many decimal places, saving with BPUT is best; but for smaller integer values, PRINT and WRITE are best.

The real number arrays of coordinates are stored with BPUT; the smaller integer values of the edge and plane arrays are stored with WRITE.

2.10.1 The Storage Routine

The storage routine is written as follows. If an object has been defined, a file select box appears which will permit the path to be stored.

If a valid file name was selected, the dimension details are stored, then the arrays *x()*, *y()* and *z()* with BPUT. Then the remaining arrays are stored with WRITE.

```
PROCEDURE storing
 '
 ' Only if an object is available.
 IF points%>0
  '
  ' Fileselector with path defined
  FILESELECT path$,"",filename$
  DEFMOUSE 5
  @get_path
  '
  ' If CANCEL was not selected, store object
  '
  IF filename$<>""
   '
   ' Open FIle and store Object information
   '
   OPEN "O",#1,filename$
   WRITE #1,points%,edges%,planes%
   '
   ' Store point coordinates
   '
   number%=points%*6
   BPUT #1,LPEEK(ARRPTR(x()))+10,number%
   BPUT #1,LPEEK(ARRPTR(y()))+10,number%
   BPUT #1,LPEEK(ARRPTR(z()))+10,number%
   '
   ' Store edge list as ASCII data
   '
   IF edges%>0
    FOR count%=1 TO edges%
     PRINT #1,kb%(count%);CHR$(10);
     PRINT #1,ke%(count%);CHR$(10);
    NEXT count%
   ENDIF
   '
   ' Store plane list as ASCII data
   IF planes%>0
    FOR count%=1 TO planes%
     PRINT #1,fl%(count%,0);CHR$(10);
     FOR j%=1 TO fl%(count%,0)
      PRINT #1,fl%(count%,j%);CHR$(10);
     NEXT j%
    NEXT count%
   ENDIF
   '
   CLOSE #1
   '
  ENDIF
 ELSE
  ' If points = 0;
  ALERT 1,"No object defined.",1," Return ",a%
  DEFMOUSE 5
 ENDIF
RETURN
```

The next procedure stores the adjusted search path, searches for the file name following the backlash. Notice that clicking on the 'Ok' button without selecting a file name returns only a backlash.

If the backlash has been found, only a *.* has to be stored. Of course, a specific extension type could be defined, for instance, *.OBJ.

```
PROCEDURE get_path
 '
 ' This procedure stores the path name
 ' selected with the File selector.
 '
 IF filename$<>""
  path$=filename$
  IF RIGHT$(path$)<>"\"
   FOR count%=LEN(filename$) DOWNTO 1
    path$=LEFT$(path$,LEN(path$)-1)
    EXIT IF RIGHT$(path$)="\"
   NEXT count%
   path$=path$+"*.*"
  ENDIF
 ENDIF
RETURN
```

2.10.2 The Loading Routine

The loading routine is determined by the way the storage of the storing routine. Corresponding to the top, the coordinate arrays are read first with BGET and then the edge and plane arrays with INPUT.

If the program has previously checked that a file with the selected name exists and if the object is not too large for the available memory area, a check for the type of file is not made.

```
PROCEDURE d_load
 '
 ' Fileselector box with path name
 '
 FILESELECT path$,"",filename$
 DEFMOUSE 5
 @get_path
 '
 ' If Cancel button was not selected and it
 ' the file exists, then load the file.
 '
 IF filename$<>""
  IF EXIST(filename$)
   '
   ' Open file and read size of object
   '
   OPEN "I",#1,filename$
   INPUT #1,points%,edges%,planes%
   '
   ' inform the user if the file is too large,
   ' then close file, and clear dimension
   ' values. Otherwise, continue loading.
   '
   IF points%>max_points% OR edges%>max_edges%
   OR planes%>max_planes%
    ALERT 1,"This file is too large|for
    available memory.",1," Return ",a%
    DEFMOUSE 5
    CLOSE #1
    points%=0
    edges%=0
    planes%=0
   ELSE
  '
  ' Load point coordinate list into memory.
  ' Array dimensions and data and zero
  ' element are bypassed.
  ' 6 bytes are necessary for floating
  ' point numbers.
  '
  number%=points%*6
  BGET #1,LPEEK(ARRPTR(x()))+10,number%
  BGET #1,LPEEK(ARRPTR(y()))+10,number%
  BGET #1,LPEEK(ARRPTR(z()))+10,number%
  '
  ' Load edge list as ASCII data
  '
  IF edges%>0
   FOR count%=1 TO edges%
    INPUT #1,kb%(count%),ke%(count%)
   NEXT count%
  ENDIF
  '
  ' Load plane list as ASCII data
  IF planes%>0
   FOR count%=1 TO planes%
    INPUT #1,fl%(count%,0)
    FOR j%=1 TO fl%(count%,0)
     INPUT #1,fl%(count%,j%)
    NEXT j%
   NEXT count%
  ENDIF
  '
  CLOSE #1
  @object_draw
  '
 ENDIF
ELSE
```

```
    ' If specified file doesn't exist:
    '
    ALERT 1,"File not found.",1," Return ",a%
    DEFMOUSE 5
   ENDIF
  ENDIF
RETURN
```

2.11. Other Functions

Now, let's take a look at the other procedures used in our 3-D program.

2.11.1 The Pixelmeter

The pixelmeter in the menu bar is processed by the procedure *pixelmeter*. The easiest solution would simply be to write:

```
PRINT AT(67,1);mx%
PRINT AT(75,1);my%
```

But as you can see, our procedure contains a few more commands. First the coordinates are updated only if the mouse position has been changed. This prevents a blinking mouse cursor.

Then the variable containing the coordinate to be displayed is filled with spaces. In this way when there is a transition from a three to two digit coordinate, no extra characters are left over when the coordinates are displayed. Also, the zero point of the coordinate system should not be in the upper left, but in the lower screen corner.

```
PROCEDURE pixelmeter
 IF xk%=1
   '
   ' Display Pixelmeter on Menubar
   '
   ' Strings for mouse coordinates
   '
   mx$=STR$(MOUSEX)
   my$=STR$(399-MOUSEY)
   mx$=SPACE$(3-LEN(mx$))+mx$
   my$=SPACE$(3-LEN(my$))+my$
   '
   ' New coordinates are displayed if the
   ' mouse is not in the Menubar, the mouse
   ' position has changed, or a menu has not
   ' been pulled down.
   '
   IF MOUSEY>20 AND (MOUSEX<>mx% OR MOUSEY<>my%)
   AND MENU(9)<>32
    PRINT AT(67,1);mx$
    PRINT AT(75,1);my$
   ENDIF
   '
   ' Store old Mouse cursor position
   '
   mx%=MOUSEX
   my%=MOUSEY
   '
 ENDIF
RETURN
```

2.11.2 The Print Handling Routines

The easiest way to send the screen to an attached printer is to press the <Alternate> <Help> key combination, or to use the HARDCOPY command in GFA BASIC. In both cases the Hardcopy routine of the operating system is used.

The operation system uses graphic handles which are not understood by most printers. The image will be printed horizontally in a size which was meant for wide printers. Circles are very elliptic on the printer. These are good reasons to write your own printer routines.

The listings of the two print routines which follow are short and simple. They were kept this way so you can file the graphic mode in the variable *graphmod$*. Check your printer manual to make any necessary adjustment. Only a few graphic modes print circles as circles; usually these are plotter and screen modes. By the way, it is a good idea to create an option for programs you want to give to somebody else that can be adjusted with *graphmod$*.

The variable *feed$* contains the size of the paperfeed between the rows of the graphic print out. Usually this value should be 24. Should you have dark shadows on the print out, the value 24 must be increased; if you have light shadows, decrease the value.

To address the growing number of owners of 24 pin printers, I added a routine which works with all 24 pin printers. Three bytes are sent to the printer for each character (3 Byte = 3 times 8 Bits = 24 Bits or needles).Since the screen is 80 Bytes wide, the last two screen-bytes in each row are padded with a null-byte (no needle) when sent.

The basic principle is to read the screen byte wise and to send these bytes to the printer. You can use the following listing to determine how the screen is going to be read.

```
FOR count%=0 to 79
  j%=32000
  REPEAT
    SUB j%,80
    POKE XBIOS(3)+count%+j%,255
  UNTIL j%=0
NEXT count%
```

Higher quality print outs are possible by developing an algorithm for reading the screen in a bitwise pattern.

The program listings are written so that the procedure which does the printing determines which print handler routine is to be used. If the printer is ready, the selected routine is jumped to. GEMDOS(17) checks for the availability of the printer. This Gemdos-routine returns true (-1) if the printer is ready to receive data.

```
PROCEDURE printer
  IF XBIOS(4)<>2
    ALERT 1,"Printer available only
    for|monochrome monitor.",1," Return ",a%
  ELSE
    ALERT 2,"9 or 24 pin printer?",1," 9 | 24 |
    Return ",a%
    IF a%<3 AND NOT GEMDOS(17)
      ALERT 1,"Printer is not ready.",1," Return
    ",a%
    ELSE
      GET 0,0,639,18,menubar$
      FOR count%=XBIOS(3) TO XBIOS(3)+1436 STEP 4
        LPOKE count%,0
      NEXT count%
      HIDEM
      ON a% GOSUB nine_pin,twentyfour_pin
      SHOWM
    PUT 0,0,menubar$
    ENDIF
  ENDIF
RETURN
```

The two called routines, which you should add to your subroutine collection, are as follows:

```
PROCEDURE nine_pin
 '
 graphmod$=CHR$(27)+"K"
 num_data$=CHR$(144)+CHR$(1)
 feed$=CHR$(27)+"J"+CHR$(23)+CHR$(13)
 '
 OPEN "",#2,"LST:"
 FOR count%=0 TO 79
  PRINT #2,graphmod$+num_data$;
  j%=32000
  REPEAT
   SUB j%,80
   PRINT #2,CHR$(PEEK(XBIOS(3)+count%+j%));
  UNTIL j%=0
  PRINT #2,feed$;
 NEXT count%
 CLOSE #2
 LPRINT
RETURN
```

The following procedure is for 24 pin printers.

```
PROCEDURE twentyfour_pin
 '
 graphmod$=CHR$(27)+"*"+CHR$(39)
 num_data$=CHR$(144)+CHR$(1)
 feed$=CHR$(27)+"J"+CHR$(23)+CHR$(13)
 '
 OPEN "O",#2,"LST:"
 PRINT #2,CHR$(13)
 FOR count%=0 TO 75 STEP 3
  PRINT #2,graphmod$+num_data$;
  FOR j%=399 DOWNTO 0
   const%=j%*80+XBIOS(3)+count%
   PRINT #2,CHR$(PEEK(const%+0));
   PRINT #2,CHR$(PEEK(const%+1));
   PRINT #2,CHR$(PEEK(const%+2));
  NEXT j%
  PRINT #2,feed$;
 NEXT count%
 '
 PRINT #2,graphmod$+num_data$;
 FOR j%=399 DOWNTO 0
  const%=j%*80+XBIOS(3)
  PRINT #2,CHR$(PEEK(const%+78));
  PRINT #2,CHR$(PEEK(const%+79));
  PRINT #2,CHR$(0);
 NEXT j%
 CLOSE #2
 LPRINT
RETURN
```

2.11.3 The Memory Alert

The memory alert informs the user how much memory area has already been used by the object and how much is still available. Since the output is in a of table within the alertbox, the main part of the routine consists of building the strings for the alertbox.

```
PROCEDURE memory_area
 '
 ' This table is placed in an Alert Box and
 ' consists of fours rows.
 '
 ' Build rows
 row1$=SPACE$(11)+"Free"+SPACE$(4)+"Used|"
 '
 ' Row for point memory area
 fr$=STR$(max_points%-points%)
 be$=STR$(points%)
 row2$="Points:"+SPACE$(6-LEN(fr$))+ fr$+
 SPACE$(10-LEN(be$))+be$+"|"
 '
 ' Row for edge memory area
 fr$=STR$(max_edges%-edges%)
 be$=STR$(edges%)
 row3$="Edges:"+SPACE$(7-LEN(fr$))
 +fr$+SPACE$(10-LEN(be$))+be$+"|"
```

```
  '
  ' Row for plane memory area
  fr$=STR$(max_planes%-planes%)
  be$=STR$(planes%)
  row4$="Planes:"+SPACE$(6-LEN(fr$))
  +fr$+SPACE$(10-LEN(be$))+be$+"|"
  '
  ' Now display the Alert box with 4 rows:
  ALERT 1,row1$+row2$+row3$+row4$,1," Return
  ",a%
  DEFMOUSE 5
  '
RETURN
```

2.11.4 Quit, Mode and Info

The procedure *quit* is quite simple. It simply checks that the user really does want to quit, and warns that any work will be lost. It then ends the program.

```
PROCEDURE quit
  '
  ALERT 2,"End the program.| |Any work not
  saved|will be lost.",1," OK | Return ",a%
  IF a%=1
   EDIT
  ENDIF
  DEFMOUSE 5
RETURN
```

The program lines associated with the menu entries Mode and Info are found in the procedure *menu_analysis*. Because so few commands are needed, an individual procedure for each is unnecessary.

2.11.5 Error Trapping

One essential item in any program is an error trapping routine. Here's a simple error handler which alerts you to any internal errors with your program and displays the number of the error. It then permits the user to select whether to continue the program, or return to the editor.

```
PROCEDURE error_handler
 '
 alrt$="Program Internal Error!|Error #
 "+STR$(ERR)+"|Save File?"
 ALERT 3,alrt$,1," Yes | No ",a%
 IF a%=1
  @storing
 ENDIF
 ALERT 2,"End Program?",1," Yes | No ",a%
 IF a%=1
  EDIT
 ENDIF
 ON ERROR GOSUB error_handler
 DEFMOUSE 5
 RESUME NEXT
 '
RETURN
```

2.12 Perspectives

So, this small 3D-program is completed. We hope, that you enjoy using it. Here's a brief explanation of some of the things you can do with the 3-D Editor.

2.12.1 Using CAD

One very significant field of computer programming is CAD (Computer Assisted Design), providing support for design and construction problems. Often, the appearance of the representation is not as important as the ability to resize it to the proper dimensions, correct lettering, and so forth.

If you'd like to pursue programming in the CAD field, try reading mathematic books, books dealing with computer-graphics, books about technical drawings, and books about 'professional' CAD Systems.

2.12.2. 3D-Drawing Programs

A second fascinating area is the creation of three dimensional drawings. Computer graphics make possible the creation of pictures which are almost impossible to draw with other technics. These beautiful pictures hide very complex mathematical formulas and a lot of calculation time.

If you are interested in these areas, you can find a lot of information in the newer literature about computer graphics. Look for terms like 'ray tracing' and 'texture mapping'. The basis of these procedures are hidden line algorithms.

You also might find *GFA Raytrace* interesting. This program is available from MICHTRON, Inc.

2.12.3 Double Buffered Applications

Three dimensional objects can also be rotated while in motion. There are two completely different methods available. One method consists of calculating during movement, the other possibility is to calculate all pictures first, without regard of the calculation time, and then display the pictures quickly, one after the other.

On the Atari ST the first possibility can only be done in Assembly Language. The advantage of this method is that one can intervene interactively in the movement of objects. Unfortunately, this can only be done at an acceptable speed with the wire-lattice models or with very small objects. This method can rarely be used from GFA BASIC, unless you use the assembler routines of *GFA VECTOR*, or your own assembly language routines.

The second possibility, double buffering, does not permit interactive control. But the advantage of this method is that the pictures can be as complex as you like. The limitations are not in the calculating capacity, but in the memory demands.

Only 32 screen pages fit into a megabyte of memory, which does not permit a long 'film' duration; therefore such operations are mostly represented in form of an endless loop. Of course, the number of pictures can be increased, by having the picture cover only a part of the screen or by using a compromised format which would not be good for the calculation time.

Text Editor

3. Writing a Text Editor

In this chapter, we will write a text editor. A text editor is a program which can be used to create files of ASCII characters. The main use for such an editor is in preparing source code (text) for a compiler language such as C or Pascal, or to write short notes to friends.

(Editors Note: You may find this explanation rather confusing. We strongly urge that you contact MICHTRON and purchase a copy of the optional disk for this book, rather than typing in the program listings. This section explains the major routines, and offers the skeleton for a text editor program. For the sake of brevity, the author choose not to explain every routine in the book text, although they are explained with comments on the disk. Please refer to the Disk Coupon at the back of this book for more information, or call MichTron.)

3.1 What should a Text Editor be able to do?

First it needs to be determined what functions the finished editor should have. Some functions are, of course, very fundamental. There must be a way to process text with the standard editor keys. Delete and backspace functions must be present, the cursor must be able to move around in the text, lines should be inserted and deleted, text should be loaded, stored, and printed.

In order to use an editor, search and replace functions must be able to be used. It should be possible to select between insert and overwrite entry modes. You should be able to move through a document a page at a time.

A very important group of functions are the block operations. There must be a way to mark areas of text and to move these blocks. Block functions should also support groups of commands such as those used from within the Block function of the GFA Basic editor.

The editor should be constructed in a manner that additional functions can be easily integrated and that any extension of the text processor is possible.

This editor will be constructed to be functional. Nothing fancy will be added. Only a white screen with a cursor will appear after the start of the program.

The next sections are as follows: Section 3.2 is a discussion of the problems which can arise while writing a text editor and how they can be solved. Section 3.3 determines a structure for the program, the function of the main program and the subroutines. How the text editor manages the text internally is explained as well as how to keep it in memory. This is done by using several arrays whose functions will be discussed.

Beginning with Section 3.4, and in the following sections, the program is developed part by part and put together.

Perhaps a word of advice on how you should read this chapter. I think it is possible to read the first parts of this chapter without trying anything out on the computer and still be able to gain an understanding. Later, it is better to take a look at the listings in the finished program and to experiment with it. Try making some changes.

3.2 Problems associated with writing a Text Editor

Text editors belong to an unpleasant group of programs from the viewpoint of the programmer. Usually they look inconspicuous and obvious, but are actually more difficult to program than a pixel oriented drawing program.

Programming a text editor requires a lot of little things which invite mistakes. Most the time these mistakes are not as obvious as with other types of programs and usually are detected too late. Especially when they only occur with certain combinations of events.

The third reason text editor programming is unattractive, is that many of the problems which arise, can usually only be solved by acquiring a better knowledge of the Atari ST and the GFA BASIC.

Some word processors and text editors on the ST, especially earlier versions, show the phenomena of key input buffering. Hold a key down, for instance during screen scrolling, then release it, and the text will continue scrolling on the monitor for some time.

We'll discuss such problems and one possible solution will be introduced.

3.2.1 Fast Scrolling with BMOVE

Everybody, who has seen text processing on the ST, knows the problem. The cursor is moved to the lower corner of the screen in order to go further down to other parts of the text, and what happens? Text begins scrolling slowly over the screen; for longer text sections, this is completely unsuitable.

Now you could understandably object, that during scrolling a word processor has more to do than our text editor. It must watch for different kinds of fonts, and so forth. This is correct, but unfortunately, this objection also affects some simple editors.

If you have purchased the optional program disk with this book, start TED.PRG from the disk and load a longer ASCII file, i.e. a *.LST-file. (Press <F1> and follow the prompts.)

Two types of scrolling are available. the slower one uses the cursor keys, the faster one the right mouse button. Move the cursor to the upper or lower corner of the screen, and press the right mouse button to try this out.

Let's analyze the problem. What should happen during scrolling? To simplify things, only look at upward scrolling, where the cursor is going past the lower screen boundary. Everything on the screen moves one line up (and the first line disappears), then a new line is displayed in the last screen row.

One idea of a solution would be to move this screen area with the GET command and to move the screen data 16 pixels higher with the PUT command. (This is the height of a text line on the monochrome monitor.)

This would be good a method, if we intend to provide boundaries on the left and/or right of our text, for example a scroll bar. But because we do not have anything on our screen besides text, we can make it scroll even faster.

To understand this method, we must introduce two things; first the construction of screen memory, and secondly the function of the BMOVE command.

Screen memory is part of the RAM. Each set bit is equivalent to one screen point. A bit is the smallest storage element of a computer, a zero or one can be stored. With the monochrome monitor, a one sets the screen point to black, a zero to white. Of course, the color settings vary with palette settings.

The ST, in high resolution mode, has a screen size of 640 pixels (picture elements) times 400 pixels, or 256000 screen pixels. Since each eight bit segment is constructed as one byte, screen memory is 32000 bytes long (this is independent of the resolution.

Since GFA BASIC has the POKE command, which allows writing a value directly to memory, you could POKE directly onto the screen, if you know where it is. Luckily, the ST has a large bag of tricks which can be used advantageously, the VT-52, BIOS, XBIOS, GEMDOS, VDI, and AES routines. If you want to do serious programming, you should obtain a reference book for these routines and learn how to use them. The book by the author of GFA BASIC, Frank Ostrowski, would be ideal, because it deals mainly with these routines and their usage in GFA BASIC. Another source of useful information is *The GFA BASIC Programmers Reference Guide, Vol. I*, by George Miller. In the next section we'll take a look at some of the VT-52 functions. We are going to use them intensively.

Let's return to the question: Where do we find the 32000 bytes of the screen memory? An XBIOS, XBIOS(2), routine returns the address of screen memory. Also, if screen memory was not changed by XBIOS 5, XBIOS(3) will provide the screen memory address. If you want to color the first point of screen memory black (on a monochrome monitor), the command is POKE XBIOS(2), 128 (128 is the value of the highest bit). Since this is hard to see (in the upper left corner of the screen), set several points near the center of the screen with POKE XBIOS(2)+17000,255.

If you play around a little bit with these addresses, you will see how the screen storage is constructed. Each one pixel high screen line can contain 80 bytes. The address of these bytes of screen memory increase from left to right.

The sequence of the lines is from top to bottom. The address of the first byte in the tenth line can be determined by:

```
9*80=720 bytes
```

XBIOS(2)+720 would be the address of the first byte in the tenth line of pixels on the screen. This byte can be filled with POKE XBIOS(2)+720,255 and the screen will display a small black line, 8 pixels long, at that point.

Let's take a small detour into the land of graphics, and see what we can do with this information about screen memory and the BMOVE command.

Find a picture in high resolution DEGAS format (.PI3) from your picture collection, or, if you have purchased the optional disk, use the sample picture on the disk. It should not contain any large white spots, to improve the effect. In the following short listing *bn$* (the name of picture) will be the name of your picture. Change the line containing *bn$* to contain the name of your picture. This program loads your picture into a previously reserved memory area and then BMOVES it to the screen. It creates a 'blind-effect', meaning the picture is moved in as though a blind were closing.

Program 3-1 A Simple Picture Loader

```
bn$="picture.pi3"
' Insert the name of your picture
'
' Reserve memory for loading the picture
'
DIM dummy$(32000/4)
'
' Find picture address in memory
'
pic_addr%=LPEEK(ARRPTR(dummy$()))
'
' BLOAD picture to this address
'
BLOAD bn$,pic_addr%
'
' Now, display the picture
' by moving it into screen memory.
'
FOR j%=0 TO 3120 STEP 80
 FOR d%=0 TO 28800 STEP 3200
   '
   ' XBIOS(2)-34 is the screen address -34 bytes
   ' which are used by the DEGAS format
   ' for information which we are not
   ' concerned with.
   '
   BMOVE pic_addr%+d%+j%,XBIOS(2)-34+d%+j%,80
 NEXT d%
 PAUSE 1  !Just to slow things down a little
NEXT j%
'
' Wait for a keypress
'
VOID INP(2)
```

Let's take a look at the BMOVE command. The 'B' stands for block, meaning a memory block or area. The 'MOVE' stands for moving, meaning that this command can move a memory area. This description is not completely correct, nevertheless it is still found in many books about GFA BASIC and even in the manual.

A more correct description would be that a COPY of a memory area is put at a different location in memory; the command should actually be BCOPY, the original area of memory remains unchanged.

The three parameters of the command are:

1) The source address

2) The destination address

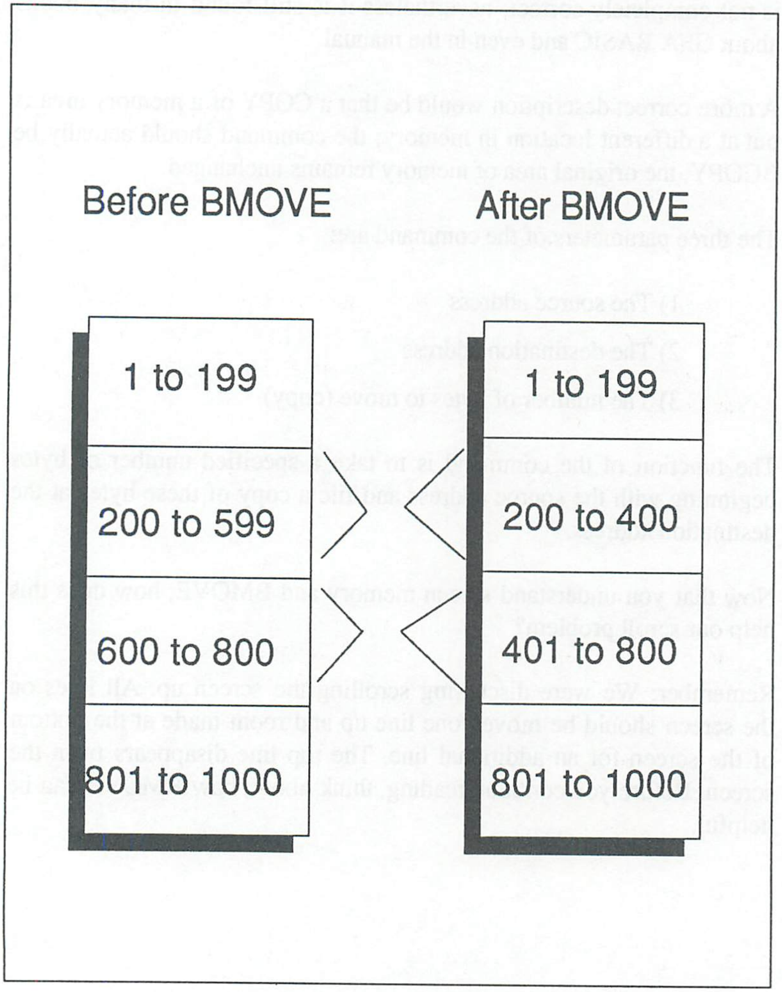
3) The number of bytes to move (copy)

The function of the command is to take a specified number of bytes beginning with the source address and file a copy of these bytes at the destination address.

Now that you understand screen memory and BMOVE, how does this help our scroll problem?

Remember: We were discussing scrolling the screen up. All lines on the screen should be moved one line up and room made at the bottom of the screen for an additional line. The top line disappears from the screen. Before you continue reading, think about, how BMOVE can be helpful.

The solution to this problem can be illustrated by this drawing:



The screen memory area starting with the second line and ending with the last position in the last line, is moved up by one line. Expressed in the parameters of BMOVE: the length of the line on the screen (80 bytes) times the height of a line (16 pixels in high resolution) plus the address of the start of screen memory.

```
16*80+XBIOS(2)
```

The command reads like this:

```
BMOVE XBIOS(2)+1280,XBIOS(2),30720
```

The first screen line is automatically overwritten. Now the last screen line must be removed and the new line written in its place. These three commands make scrolling of a line possible. This method of scrolling is rather fast.

Now that moving lines on the screen has been explained, the output of a new line on the bottom of the screen per print command is no problem either, but how is the last screen line deleted?

The obvious (and slow) method would be simply to create an eighty character long empty string, with SPACE$(80), and to write that on the last line.

But there is another possibility which is better for three reasons: First: it is faster (about 7 to 8 times), secondly: it is instructive, because it highlights a nice, often ignored, area of the ST, and third: it is a good bridge to the next section.

3.2.2 The VT-52 Sequences

The Atari ST has several different commands which concern the screen handling. These commands are known as the VT-52 escape sequences. They are called escape sequences, because they consist of an escape character (ASCII-code 27) and a letter. Some of these commands need an additional parameter, too. The term 'VT-52' is in reference to a type of terminal.

These commands may be used to delete the screen, to insert a line from the cursor position, to turn on or off reverse character, and other functions. We won't list all of the VT-52 commands, but only those which we can use in our editor.

The command Escape l is responsible for deleting a line, the task we discussed in the last section. In order to use this command from GFA BASIC, the ASCII code for Escape, (27) and the ASCII code for the lowercase 'l' are sent to the screen by a PRINT command. The command to delete a line is then:

```
PRINT CHR$(27)+"l";
```

or

```
PRINT CHR$(27);CHR$(108);
```

The semicolon at the end of the line prevents the cursor from being positioned on the next line (it suppressed the linefeed). If the cursor were in the last line, which is the case in our example, the screen will scroll one line up. Try this out now by putting in the following lines:

```
PRINT AT(1,24);"Testline";
PRINT AT(1,25);"Deleteline";
VOID INP(2)   ! wait for keypress
PRINT CHR$(27)+"l"
VOID INP(2)
```

You will find out, that the line containing 'deleteline' will be deleted, but the line with 'testline' will move one line higher on the screen. However, if you end the last command with a semicolon, 'testline' will remain in it's old position.

A remark about VOID INP(2): the INP command returns one byte from a specified source. The 2 stands for keyboard; INP(2) is waiting for keypress. But what does VOID stand for?

The command INP(2) gets a byte from keyboard and expects to store this byte somewhere, usually in a variable. Simply writing INP(2) generates a syntax error, since there is nowhere for the byte to be stored. We could say:

```
key=INP(2)
```

and the pressed key would be stored in the variable *key*. The value of this variable is not important to us, all we do is waste memory with it. The command VOID tells the following command, to forget it's return value. Even arithmetic commands can be done with VOID, like:

VOID SIN(X) or VOID 2+3

But this would make no sense, since the result is calculated, but immediately forgotten. The execution of a command without storing the return value is faster and this can be helpful with some applications like time critical switching between screens.

Getting back to the VT-52 sequences. In our editor we will have built in commands, to insert and delete lines. Such commands, called by pressing the respective command keys, leads to two consequences. One consequence is that a change is made to the data, the other consequence is that something is changed on the screen. The changes on the screen can be handled with the VT-52 commands.

In order to delete a line, use:

```
PRINT CHR$(27)+"M";
```

The difference between Escape-M and the previously mentioned Escape-l, is that this command not only deletes a line at the cursor position, but scrolls the rest of the screen up, so the last screen line is available.

```
PRINT CHR$(27)+"L";
```

The Escape-L sequence inserts an empty line at the cursor position and scrolls the rest of the screen down from the cursor position. The lowest screen line is scrolled off of the screen.

You can test these commands with the following short program:

```
' Print one letter on each screen line
'
FOR a%=1 TO 25
 PRINT AT(1,a%);CHR$(a%+64);
NEXT a%
'
' Wait for a keypress
VOID INP(2)
'
' Delete line with the letter "E"
' and scroll remaining lines up
'
PRINT AT(1,5);CHR$(27)+"M"
VOID INP(2)
'
' Inserts an empty line after the second line
'
PRINT AT(1,3);CHR$(27)+"L"
'
VOID INP(2)
```

Now here are some very fundamental VT-52 commands. After starting our editor, an initialization routine is called, in which some arrays are defined. Then a cursor is displayed on a blank white screen. But how do we get the cursor?

All PRINT, INPUT and other screen oriented commands use a cursor, which is invisible. Turning the cursor on and off is possible with a VT-52 command:

Turn on:	PRINT CHR$(27)+'e';
Turn off:	PRINT CHR$(27)+'f';

After starting the program the cursor must be turned on with the proper command and it must be turned off during some other operations, such as insertion of the help screen with which displays the commands of our editor.

The cursor is not supposed to just sit on one spot, it should be able to move around. This is accomplished with the VT-52 escape commands:

Cursor up:	PRINT CHR$(27)+"A";
Cursor down:	PRINT CHR$(27)+"B";
Cursor to the right:	PRINT CHR$(27)+"C";
Cursor to the left:	PRINT CHR$(27)+"D";

The command is ignored when the cursor reaches the edge of the screen. Earlier in this book we mentioned commands for turning on and off inverse text mode. To refresh your memory, these commands are:

Inverse text on:	PRINT CHR$(27)+"p";
Inverse text off:	PRINT CHR$(27)+"q";

Test these commands with the following listing:

```
PRINT "Normal"
PRINT CHR$(27)+"p";"Reverse"
PRINT CHR$(27)+"q";"Normal"
```

We should mention something else before going on to the last useful command. There are VT-52 commands to delete screens, to place cursors, and so forth, which are already used in GFA BASIC as commands (CLS and PRINT AT) and are therefore not very important for us. In addition there are commands, which can be used for scrolling text, without using the somewhat more difficult method of BMOVE from the last section. We opted for the BMOVE command, because it is faster and screen scrolling is a time critical operation which should be as done as fast as possible.

Now to the last application of a VT-52 command:

```
PRINT CHR$(27)+"w";
```

This commands turns off the automatic line overflow. This means that for a line of text with more than 80 characters being displayed with a PRINT command on the screen, all letters which do not fit into the 80 character line will be displayed on the next line. This can be prevented by turning off the automatic line overflow. Subsequent PRINT commands which will use more than one line are not possible.

Suppose you have a line with 80 text characters. This line moves into the last screen line by scrolling and is supposed to stay there. The command:

```
PRINT AT(1.25);SPACE$(75)+"Test";
```

is not enough; the line scrolls up to screen line 24. This can only be prevented, by turning off the automatic line overflow at the start of the program.

This can be demonstrated with:

```
PRINT AT(1.25);CHR$(27)+"w";SPACE$(76)+"Test";
```

In the next section we will not be discussing how to get something out of the computer, but how to get something into it.

3.2.3 Possibilities of Key Interrogation

A text editor is basically very simple. Many subroutines are available for the different functions, like delete line, copy block, load text, and so forth. These subroutines can be called by the user by pressing a key. The principal work of the editor is reading user commands from the keyboard or mouse and the selecting the proper subroutine.

Keyboard interrogation and branching to subroutines will be the topic of this section. First, the problem of keyboard interrogation. Several commands are available in GFA BASIC to check the keyboard:

```
INP(2)
INKEY$

ON MENU
```

Each one of these methods has its advantages and disadvantages.

First, let's look at the INP(2) command. INP(2) reads a byte from the console and returns the ASCII code of most keys. (It returns high ASCII codes with special keys such as function, cursor keys, delete, and home keys, which can be used to distinguish the keys.) Unfortunately, this command waits for a key to be pressed. This is therefore unsuitable for us, because it must be possible to position the cursor with the mouse. If the INP(2) command and commands for handling the mouse are placed into a loop, GFA BASIC would be hung up at INP(2) command and mouse handling waiting for a keypress.

The INKEY$ command is more suitable from this point of view. It returns a string, from which the pressed key can be evaluated. If no key was pressed, the command returns an empty string. Because this command is not waiting for a keypress, other commands can be put with it in a loop and the mouse can be watched at the same time.

As already mentioned, an empty string is returned when no key has been pressed. If a key has been pressed, this string is either one or two characters long. To understand this, a few words need to be said about the keyboard of the ST. Basically, all keys of the ST can be divided into three groups.

The first group are the keys with assigned ASCII codes. These are letters, special characters and the spacekey; all printable characters. Unfortunately, some keys with non printable characters return an ASCII code too. This code is smaller than the code of the first printable character (this is the space key, ASCII code 32).

These keys are the <Esc> key (ASCII Code 27), the <Return> key (13), the <Tab> key (9) and the <Backspace> key (8).

One key disturbs this scheme, the <Delete> key. It returns the ASCII code 127. This is defined as a small triangle. In our editor this key will have a different function. All of the keys in this group return a string with a length of one with the INKEY$ command.

No distinguishable ASCII code is assigned to the second group of keys, the returned ASCII code is always zero. The function keys, the <Insert> and <Clr/Home> keys as well as the <Help> and <Undo> keys belong to this group. They can be distinguished through the scan code rather than the ASCII code.

If one of the keys from this group is pressed, INKEY$ returns a two character string. The first character is a zero byte and the second character contains the scan code of the pressed key. You can analyze the keys with the following listing to determine the values returned by the INKEY$ command:

```
' KEY.LST Program 3-2
PRINT "Length:","First Byte:","Second
Byte:","ASCII Character:"
REPEAT
 key$=INKEY$
 IF key$<>""
  PRINT LEN(key$),ASC(key$),
  IF LEN(key$)=2
   PRINT ASC(RIGHT$(key$)),"-----"
  ELSE
   PRINT "-----",LEFT$(key$)
  ENDIF
 ENDIF
UNTIL MOUSEK
```

The third group of keys are the switching keys. These are the right and left <Shift> keys, the <Control>, the <Capslock> and the <Alternate> key. INKEY$ returns an empty string, meaning the keypress was not registered. If a switching key is pressed and another key at the same time, this can produce a different return code than the solitary press of the key. We can use this especially with control commands, when the <Control> key and a letter key are pressed at the same time.

By trying out all the switching keys, you will find a few problems after a while, especially regarding the cursor keys. Pressing the <Shift> and cursor keys in unison produces the same codes as several number keys. If you feel that this is disagreeable, you must filter it out.

Another strange point you will find in connection with the <Delete> and <Insert> keys is that the <Delete> key produces a different code than <Control> plus the <Delete> key, but <Insert> results in the same code as <Control> plus <Insert>. If you want to assign different functions to <Insert> and <Control> plus <Insert>, the keyboard switching keys must be checked again by something in addition to INKEY$.

This can be done with an operating system routine, BIOS(11). The following listing permits checking the status of the keyboard switching keys, without checking any other keys at the same time:

```
REPEAT
  switch%=BIOS(11,-1)
  IF switch%
    PRINT switch%,
  ENDIF
UNTIL MOUSEK
```

With INKEY$ and BIOS(11,-1) every key, can be checked in the same loop. While the keys are being checked, the mouse can also be handled. This seems to be ideal for our purposes. Nevertheless, there remains one last method of keyboard management which needs some discussion.

This third method is by using the ON MENU command. This command returns the value of the pressed key in several system variables. The variable *MENU(14)* returns a value of the Scan and ASCII codes, *MENU(13)* returns the status of keyboard switching keys.

Since this command as well as INKEY$ does not wait for a keypress, it seems to be as useful for our purposes as INKEY$. But if you think about what an ON MENU command does, you may have some doubts. The ON MENU command uses a very complex routine from the user interface of the ST, an AES routine. This routine, Evnt_multi, is probably the most complex in GEM. It waits for multiple events and is able to induce reactions to these events. At the same time, it watches over the keyboard, Pull Down Menus, the mouse and some other events, too. The number of parameters, which this function needs from the programming language in C, shows its complexibility; there are 24 input and return parameters. GFA BASIC programmers using Version 3.0 may use the GFA BASIC command EVNT_MULTI (refer to your manual), but this command is only available as a GEMSYS call from earlier versions of GFA BASIC.

The number of parameters indicates that this function is a bit too large for our purposes and also relatively slow. A function that does a lot needs a lot of time. Therefore, we are going to set this function aside and handle the keyboard with INKEY$, eventually supplemented with BIOS(11,-1).

That solves the first part of our problem. Our editor now can detect which key has been pressed; but how does it select the program subroutine which shall be called?

The normal method would be a command, which could react to several expressions of a variable, a multiple branching command. This can be the SELECT/CASE command of GFA BASIC 3.0, but such a command does not exist in GFA BASIC 2.0. In order to make this program useful to as many programmers as possible, we'll stick with commands which can be used in both versions of GFA BASIC. Programmers with Version 3.0 might be interested in experimenting with this function.

An ON variable GOSUB command exists, but this command expects the variable to contain values between 1 and the number of procedures which follow. Therefore it, too, is not suitable. What can be done?

A rather inelegant solution is to write an IF THEN statement for each subroutine to check for the appropriate keypress.

A better solution is to check the Scan string. Here's an example which will illustrate this method.

If you were to write a program which displays a menu of the first letter of each command on the screen, you might use the following logic:.

To get the keypressed for the command, you would use:

```
letter$=CHR$(INP(2))
```

Now it would be inelegant to write:

```
IF letter$="T"
 GOSUB textinfo
ENDIF
IF letter$="D"
 GOSUB datainfo
ENDIF
' and so on...
```

When you have a large number of menu items to check, this program construction gets real boring.

Instead of this, you could write:

```
scanstring$="TDGPH"   ! the possible menu
selections
ON INSTR(scanstring$,letter$) GOSUB textinfo,
datainfo, etc...
```

INSTR(scanstring$,letter$) finds which position is held by the pressed letter key in the string *scanstring$*. This determines which procedure in the following procedure list shall be branched to. Of course, the pressed key must correspond to a letter in *scanstring$*.

Even in our editor, we can create a *scanstring$* with the possible keys and branch to them as shown in this example. Since the procedure list will soon get too long, several ON...GOSUB statements can be used.

We'll be using the awkward solution of IF THEN, because it is somewhat clearer and can also be expanded easily by additional key commands. The editor program checks for a keypress, if the received string returned by INKEY$ is one or two characters long, further evaluation is required. If it is two characters long, the procedure *command_2* takes over. If it is only one character long, the ASCII code is smaller than 32 or equal 127 (delete), the evaluation is done in procedure *command_1,* otherwise the string is a printable character and is processed in the procedure *printable_character*.

There are some problems with this method, too. For each key value returned with a length of two, more than 20 IF conditions must be tested, and only one can be correct. Even if the first one is the correct one, all of the following conditions are still checked. This is not tragic with most commands because their call is not time critical, but with other some other time critical functions, this delay can be bothersome.

Unfortunately, it is not possible in GFA BASIC to write a RETURN before the ENDIF's of the IF conditions in order to leave the procedure. Only one RETURN can belong to a procedure. But this can be done, by putting a label before the RETURN, and then jumping to the label with a GOTO. Usually GOTO is a sign of a bad programming style, but in this case the GOTO command is permissible. It results in breaking up the procedure, and it is tested by the preceding IF condition. It effects the procedure the same as an EXIT IF statement. It would not make sense to supply each subroutine block with a GOTO *label*. This is only necessary with some of the time critical functions, for instance scrolling and turning pages.

With the problems of handling the keyboard and branching to subroutines solved, and we can move on to the problems associated with block operations.

3.2.4 Fast Block Functions

Most text processors and text editors use block operations which move, copy, delete, and perform other operations on lines of text. These operations are usually pretty fast, so you will probably not suspect any problems.

Let's look at the simplest of all possible data structures for a text editor. In this structure each line of text is held in a string array, *line$()*.

Suppose the text is 1000 lines long and lines 600 to 800 needed to be moved to position 200. This results in the following changes to the text: the present lines 200 to 599 turn into lines 401 to 800 and the present lines 600 to 800 into 200 to 400. The following chart should help explain this principle.

Before the move	After the move
1 to 199	1 to 199
200 to 400	401 to 601
401 to 599	602 to 800
600 to 800	200 to 400
801 to 1000	801 to 1000

These moves must be done line by line with statements like:

```
line$(c)=line$(j)
```

The required time would be enormous. The problem would be even clearer, if you deleted line 5 in the text. With 2000 lines you would have to write:

```
FOR c%=5 to 1999
 line$(c%)=line$(c%+1)
NEXT c%
```

Compiled GFA BASIC will need about a third of a second for this operation. The editor we'll be constructing will use about a sixth of the time using a different process.

The solution of the problem consists mainly in the selection of a suitable data structure. The lines of the editor can be in a string array, but they must be managed with additional arrays. What these arrays look like and how they are manipulated, are concept dependent.

Let's look at two different programming concepts. One is the concept of interlinked lists. Two arrays exist. One contains the text lines. The sequence of lines in this array is whatever you choose, it does not have to be in accordance with the sequence of lines in the text. The other array contains pointers to the text lines. In our case, the pointers point to the lines in the string array and describe the sequence of the lines in the text.

The zero element of the pointer array describes which line is first. You could also say, the first line is stored in *line$(pointer%(0))*. The next line can only be found by using the pointer If the value of the pointer is five, that means the next line is line 5. The pointer is the index to the next line.

Here's an example to make this principle clearer. The following text will be displayed on the screen:

aaa
bbb
ccc
ddd
eee

But in the line array, the text is stored as::

```
line$(1)="ccc"
line$(2)="aaa"
line$(3)="eee"
line$(4)="bbb"
line$(5)="ddd"
```

The pointer for the interlinked list looks like this:

Number	Pointer	String
0	2	<not available>
1	5	ccc
2	4	aaa
3	NIL	eee
4	1	bbb
5	3	ddd

'NIL' means Not In List. This is how the end of the list is marked. This is represented in the pointer structure by a number which can not point to a string array, -1.

Pointers may contain the array index of the next line and of the next pointer. Usually pointers do not point at indexes, but at addresses in memory. Don't be surprised if you find different conditions for pointers in C or Assembly Language or even with GFA BASIC pointers using asterisk.

Perhaps we should discuss an index field in connection with our intended usage. To display the five lines shown above on the screen, we could use the following routine:

```
'LIST.BAS
DIM line$(5),pointer%(5)
'
FOR cnt%=0 to 5
 READ pointer%(cnt%),line$(cnt%)
NEXT cnt%
DATA 2,,5,ccc,4,aaa,-1,eee,1,bbb,3,ddd
'
index%=0
REPEAT
 PRINT line$(pointer%(index%))
 index%=pointer%(index%)
UNTIL index%=1
```

The first loop reads the items in the data statement into a table. Each pointer is associated with a string array. The string of the zero element is not needed. The variable *index%* starts with a value of zero, and *pointer%(index%)* is equal to two. The first line 'aaa' is found in *line$(2)*, *pointer%(2)* is the number of the next line and so on.

Experiment with this table. Most people have some problems when confronted by an interlinked list for the first time. The output loop uses the pointer to the interlinked list.

You're probably wondering why this system is important. Imagine that a line must be deleted. For example, delete the third line in the preceding example. All that needs to be done is to change the pointer. The pointer of the preceding line is simply changed to the value of the pointer to the line which follows the line to be deleted.

This does not change, even if you reach for the more liked and somewhat more flexible procedure of the double interlinked list. Two pointer arrays exist with that. One points to the follower of the actual element (like before), and the other one points to the predecessor; one can go hand over hand in both directions of the list.

The five lines with their pointer arrays would read like this in a double interlinked list:

Number	Follower	Predecessor	String
0	2	<not used>	<not available>
1	5	4	ccc
2	4	NIL	aaa
3	NIL	5	eee
4	1	2	bbb
5	3	1	ddd

In the unused first element the predecessor pointer could point to the last element of the list, then this one would be equal to 3.

If you understand the procedure, think about which operations are necessary for block movements as mentioned in the beginning of this section. The solution is very elegant, but also very tricky.

We're not going to use this method of text management in our editor. We will use a simpler one, with two arrays. In one of the arrays, the lines are stored in strings. The other array is made up of pointers. The pointer points to the line in the text, there is no interlinkage. The pointer system for the layout of the lines in the string array would then be:

Number	Pointer	String
1	2	ccc
2	4	aaa
3	1	eee
4	5	bbb
5	3	ddd

The first line 'aaa' is found in the second item of the line array, therefore is the variable *pointer%(1)=2*. Displaying the five lines is done with this simple loop:

```
FOR cnt%=1 TO 5
 PRINT line$(pointer%(cnt%))
NEXT cnt%
```

It's also possible to jump directly to the middle of the text. The third line, for example, can be reached by simply using *line$(pointer%(3))* without going item by item through the array.

Remember the problem with deleting a line? Suppose, the fifth line is not to be deleted, but a new line inserted as the fifth one. All lines, including the fifth line, will be moved up one line. The slowest solution would be:

```
FOR cnt%=2000 DOWNTO 5
 line$(cnt%+1)=line$(cnt%)
NEXT cnt%
line$(5)=""
```

But how is the problem to be solved with the suggested data structure? One apparently foolish method would be:

```
FOR cnt%=2000 DOWNTO 5
 pointer%(cnt%+1)=pointer%(cnt%)
NEXT cnt%
pointer%(5)=2001
' The new line goes at the end of
' the line storage, in line$(2001).
```

This method is rather cumbersome, however, the method we are going to suggest really follows this path. But, counting the loop commands, instead of executing almost 6000, we'll use only 2. If we want to write so that it's easy to follow, we'll use 6 commands. The result will be exactly the same.

Each element of the pointer array between the sixth and two thousandth is equal to it's predecessor (*pointer%(cnt%+1)=pointer%(cnt%)*) and the fiftieth element is equal to the end of the list.

To explain how this functions, you must understand the Bmove command and a few things about numerical .i.arrays;. The Bmove command was discussed at length in the section on fast scrolling. If you are unsure how it works, review that section.

We're only interested in filling the numerical arrays. String array management is a little more complicated.

Each numerical array has a descriptor. This descriptor is 6 bytes long and contains information about the array. The first four bytes contain the address of the array. The remaining two bytes contain the number of dimensions of the array. Those two bytes are of no interest to us, since we will only use one dimensional fields.

The address of the data (in memory) can be determined by using the array descriptor. The address of the descriptor is returned by:

```
addr=ARRPTR(variable_name())
```

Since the first four bytes are the address of the array, Lpeek(Arrptr(variable_name())) returns the starting address of the data array.

Beginning at this address are four bytes which state the number of elements the array can hold. The data in the array follows. Floating point arrays require six bytes for each array element, integer fields only require six bytes. The address returned by Lpeek(Arrptr(pointer%())+4) would contain the value of *pointer%(0)* (assuming that OPTION BASE 0 is being used).

The command OPTION BASE (0 or 1) can be used to determine if there should be an array index of zero, or if the array should begin with element number one. If OPTION BASE 1 has been selected, then the address will contain the value of *pointer%(1)*.

The address of an array element can be also be determined by using the command VARPTR(). This is especially useful with one dimensional arrays and OPTION BASE 0:

```
LPEEK(ARRPTR(x%())+4)=VARPTR(x%(0))
```

The command sequence to insert a line at line 5 would be:

```
FOR cnt%=2000 DOWNTO 5
 pointer%(cnt%+1)=pointer%(cnt%)
NEXT cnt%
pointer%(5)=2001
```

This will move all pointers from the fifth (inclusive) one position higher and put a new value into *pointer%(5)*

We could also determine the address of *pointer%(5)* and move the data of the 1996 pointers to be moved up by 4 bytes with the BMOVE command.So instead of using

```
FOR cnt%=2000 DOWNTO 5
 pointer%(cnt%+1)=pointer%(cnt%)
NEXT cnt%
```

we can use:

```
base%=LPEEK(ARRPTR(pointer%()))
source_address%=base%+5*4
dest_address%=source_address%+4
number%=1996*4
BMOVE source_address%,dest_address%,number%
```

or even:

```
BMOVE VARPTR(pointer%(5)), VARPTR(pointer%(6)),
1996*4
```

This does look more difficult, but because there is no loop, it is about 85 times faster (in the Interpreter). This increase in speed makes it worthwhile. The address of the variable *base%* only needs to be found once at the beginning of the program.

As you can see, the address for the start of the BMOVE can be found by using ARRPTR as well as VARPTR. When ARRPTR is used, you must still calculate the distance of the address from the start of the array.

We will use ARRPTR in the text editor program. It does have the disadvantage of looking more complicated than VARPTR.

The other problem mentioned earlier was block moving. To refresh your memory: the text is 1000 lines long. The block starts at line 600 and ends at 800. We want to move this block to line 200. Three BMOVE operations are necessary. First, the pointers from 600 to 800 are moved to a safe temporary storage area. Then the pointers from 200 to 600 are moved upward by 200, and finally, the intermediate storage area is moved to position 200. This is done very fast, since loops are not used.

That covers everything we need to know for the block moves in our simple editor. We should warn you about one thing before going on. In 'normal' programming there is little danger of crashing (with bombs) or freezing (without bombs) in GFA BASIC. But when you begin using commands such as BMOVE, POKE and PEEK, and so forth, this does not necessarily apply.

Something as simple as a typing error, like typing 'addrees' instead of 'address' can cause serious problems. If you mistyped the name of the variable for the BMOVE, the value could equal zero, and the memory being moved will overwrite other memory areas, areas not intended for your use.

Also be careful when using BMOVE that no memory areas outside of the arrays can be reached.

Now there is only one large problem with programming a text editor left to consider, the keyboard buffer.

3.2.5 The Keyboard Buffer

Some text editors have an unfortunate tendency to cause keyboard overflow. This means that if a key is pressed for a period of time, for instance, the cursor down key, and then the key is released, the function of the pressed key continues, instead of stopping as intended.

This can be demonstrated with the following routine:

```
REPEAT
 PAUSE 5
 IF INKEY$<>""
  PRINT "Running"
 ENDIF
UNTIL MOUSEK
```

During the PAUSE command, keypresses are registered, but cannot be carried out.

The reason for this is the Keyboard Buffer. This is a buffer which stores any keypresses which the computer is not in the position to carry out immediately. The task assigned to the key, even one as simple as writing to the screen, will be carried out at a later time. This is a common problem with all computers. The ST's keyboard buffer is defined as 256 bytes long, but this length can be changed.

Of course, we want to eliminate this problem in our editor. Fortunately, there are different methods available to do this.

The keyboard buffer is managed by a table. The first four bytes of this table contain the address of the keyboard buffer. The next two bytes contain the length of the buffer. The following two bytes contain the head_pointers and the last two bytes the tail_pointers.

Two pointers manage the contents of the keyboard buffer in a way that you might find somewhat incomprehensible. Between the head_pointer and the tail_pointer the keyboard buffer holds characters, which each use four bytes.

If there is no character in the keyboard buffer, head_pointer and tail_pointer will point to the same value. If the keyboard buffer starts to fill up, the head_pointer will move away from the tail_pointer, and the characters are placed between them.

If one of the pointers is at the end of the keyboard buffer area, it will jump back to the beginning; it always runs in a circle. The following listing should be of some help in analyzing this situation.

This program needs to know where the keyboard buffer is in memory. An XBIOS routine, XBIOS(14,1) returns the address of the table.

```
' KEY_BUFF.LST Program 3-3
'
DIM tail%(20),head%(20)
'
PRINT "Enter 20 keypresses:"
FOR cnt%=1 TO 20
 a%=INP(2)
 PRINT CHR$(a%);
 PAUSE 5
 head%(cnt%)=DPEEK(XBIOS(14,1)+8)
 tail%(cnt%)=DPEEK(XBIOS(14,1)+6)
NEXT cnt%
'
CLS
PRINT "Head_pointer","Tail_pointer"
FOR cnt%=1 TO 20
 PRINT head%(cnt%),tail%(cnt%)
NEXT cnt%
'
REPEAT
UNTIL MOUSEK
```

Now start the program and hold a key down until the table appears. Take a look at the numbers under the Tail_pointer heading. They increment by steps of four, because one character was read during each pass through the loop. Each time the INP(2) statement was encountered, 4 bytes were read intò the keyboard buffer. With a little luck you may have found the spot where the pointer jumps from the end of the buffer (252) back to the beginning (0). (It may be necessary to increase the size of the loop and read more keystrokes. If you do increase the loop size, remember to increase the size of the arrays.)

The head_pointer moves in larger steps than four bytes, because several keypresses are registered and stored during each pass through the loop (because of the PAUSE statement), and only one can be processed by INP(2).

To prevent keyboard overflow in the editor, the keyboard buffer must always be emptied, after a command sequence during which no keypress analysis is possible. The following procedure was suggested in a well known German computer magazine:

```
WHILE INP?(2)
  VOID INP(2)
WEND
```

This routine checks the console with INP?(2) to see if any key has been pressed. If a key has been pressed, a value of TRUE, -1, is returned, and the character is read by INP(2). The basic idea is not bad, but it's not very elegant.

One possible side effect of this routine is that scrolling could become slower if the keyboard repeat rate is set high, because it will take more time to clear the keyboard buffer. (More characters are stored in the buffer.)

We need a more elegant solution. We saw that a keyboard buffer is considered to be empty if its head_pointer and tail_pointer contain the same value. Since we know the addresses of head_pointer and tail_pointer, we can read one value and write it at another position with:

```
DPOKE XBIOS(14,1)+6,DPEEK(XBIOS(14,1)+8)
```

Since the two by two bytes are in sequence, both can be changed with an LPOKE, where the two high bytes have the same value as the two low bytes. The simplest of these values is 0. Therefore we will clear the keyboard buffer with:

```
LPOKE XBIOS(14,1)+6.0
```

Our problem is solved.

To illustrate another point, try entering the following commands:

```
LPOKE XBIOS(14,1),XBIOS(2)
DPOKE XBIOS(14,1)+4,32000
```

Run this short program, then go into the direct mode (<Shift> + <F9>) and press and hold the cursor down key. Strange lines will begin to appear on the screen. Exit GFA BASIC and return to the desktop. Press and hold a key again. The same lines appear. What happened?

The first command line moved the keyboard buffer to the beginning of screen storage, the second line defined the buffer as the same size as screen memory (32000 bytes). This looks dangerous, but is actually quite harmless and (usually) will not lead to a crash. (It will disappear the next time your ST is reset.

Now, we have considered all of the larger problems we can anticipate in writing a text editor. It's time to think about the actual construction of the program.

3.3 Data and Program Structure

Some of the things we'll discuss in this section have been said before. We'll be looking at how our editor can be efficiently constructed. Two points must be clarified. First: How is the data stored and managed in memory? Second: What are the parts of the program able to do and how do they work together. Stated differently: What is the program supposed to do?

3.3.1 Lines, Pointer and Dummy Arrays

We've already covered fast block functions. We'll use an array called *line$()*, which will contain the text. The sequence of the lines in the array does not necessarily have to be in sequence with the lines of text.

A second array, *pointer%(cnt)*, describes, at which position in the array a line can be found. To make fast processing of the text possible, such as with block operations, as few operations as possible should be done in loops. Also, few operations should deal with the actual line storage. We'll use the pointers for those purposes.

Be sure that you understand this data structure, because the whole program is built on it. If you have any doubts, reread the explanation of this data structure in the section titled 'Fast Block Functions'.

For some operations, like moving a block, it is practical if a larger group of pointers can be stored. For this purpose, we will use a dummy array. The sole job of the dummy array is to protect a block of memory from other variables. Often those memory blocks are used to hold Assembly Language routines, or to store screen pages.

Now a restriction about line storage. This restriction is not necessary, but simplifies many operations. Only the first strings of the number array are supposed to be occupied; the empty lines should follow them. Variables 1 to 10 are never followed by a few empty variables, with the remaining lines starting at index 15. This restriction simplifies most operations, but unfortunately interferes with at least two functions (delete line and block) and costs processor time.

If you understand the editor and want to improve it; you can ignore this restriction.

3.3.2 The Main Program and Procedure Structure

As we already mentioned, we will use a short main program, which the checks the keyboard with INKEY$ and checks the mouse. By pressing a key the main program will branch to one of three subroutines, which carry out the necessary functions depending on which keys are pressed.

These three subroutines contain IF THEN statements which check which specific key has been pressed. Time critical functions are at the beginning of the procedure, before the ENDIF of the command block is a *GOTO end_of_procedure*, so no unnecessary functions are checked.

Besides the three primary subroutines, several supporting procedures are used. These procedures display the text beginning at a specific line on the screen.

Finally, we have completed our discussion concerning the theoretical part of our text editor and are prepared to start the actual job of programming the computer.

To get the most value from this program, try to change and improve it. Make sure that you make your changes to a copy, and not the original version of the program. Then, if something doesn't work out as planned, you can back up and start fresh.

The following sections each explain a subroutine. The construction of the program is explained first and the listing follows each section. The program listings contain remarks to clarify some points.

3.4 The Initialization Part

When the program starts, a check is made for the proper screen resolution. (This program was written for high resolution, if you are using a color monitor, you'll have to make a few changes, as explained in previous sections of this book..) The resolution can be found by using the XBIOS(4) function, which returns the following values:

0	Low resolution (320x200)
1	Medium resolution (640x200)
2	High resolution (640x400)
3	Reserved

Then a procedure called 'preparation' is called, which does the initialization of the program; arrays are defined, a error handling routine is called, and so forth. Two commands need to be examined a little closer. OPTION BASE 1 defines that array index, the array index starts at one. This will save some memory, but this is not very important in this case.

The line containing VOID XBIOS(21,3,-1) turns off the cursor blink. Unfortunately there is a small incapability between interpreter and compiler, the cursor blinking in the interpreter has to be turned off. The -1 says that this parameter should not be changed. The value specifies the blink rate of the cursor, while a 3 represents a non blinking cursor.

The REPEAT loop initializes all pointers. The pointer points at the line, which may change during the program.

```
IF XBIOS(4)<>2
 ALERT 1,"Warning!| |This program was
 written|for the High Resolution|(640x400)
 Monitor",1," Return ",a%
ENDIF
@preparation
```

and:

```
PROCEDURE preparation
 '
 '
 OPTION BASE 1
 ON ERROR GOSUB error_handler
 '
 ' Reserve memory and initialize arrays
 '
 max_lines%=FRE(0)/100
 max_lines%=MIN(max_lines%,5000)
 'Lines of text:
 DIM line$(max_lines%)
 ' Pointers to text lines:
 DIM pointer%(max_lines%)
 ' Array needed by some block operations:
 DIM dummy%(max_lines%)
 base%=LPEEK(ARRPTR(pointer%()))+4
 dummy_base%=LPEEK(ARRPTR(dummy%()))+4
 '
 ' Now define some additional variables.
 ' Null settings are actually unnecessary.
 '
 insertmode!=TRUE
 cnt%=0
 REPEAT
  INC cnt%
  pointer%(cnt%)=cnt%
 UNTIL cnt%=max_lines%
 path$="\*.*"
 ' Number of the text line
 ' in the first screen line:
 upper%=0
 ' Turn off cursor blink for Compiler
 VOID XBIOS(21,3,-1)
 ' VT-52 escape code for
 ' automatic line overflow off
 PRINT CHR$(27)+"w";
 ' VT-52 escape code for show cursor
 PRINT CHR$(27)+"e";
RETURN
```

3.5 The Main Program

The main program is an endless DO loop, which checks the keyboard and the mouse. If a key is pressed the appropriate procedure is called.

The cursor may be positioned by pressing the left mouse button to call the procedure *place_cursor*. We'll discuss this in detail later.

The right mouse key permits scrolling. The text will be scrolled up if the mouse cursor is in the first five lines, down if the mouse cursor is in the last five lines. Scrolling with the mouse is much faster than scrolling with the keyboard.

If you examine the listing, you will see that instead the two scrolling procedures, other command sequences for scrolling with the mouse are used. These commands represent a scrolling routine adjusted to the mouse, which is about 20% faster than the keyboard scrolling procedures. However, the function is exactly the same.

```
DO
 '
 ' *** Key analysis ***
 '
 key$=INKEY$
 asc%=ASC(key$)
 '
 ' Cursor control, Help, Undo key, etc.
 '
 IF LEN(key$)=2
  @command_2
  '
  ' Clear keyboard buffer
  ' Set head and tail pointers to same value.
  '
  LPOKE XBIOS(14,1)+6,0
  '
 ENDIF
```

```
'
' Check for printable keys
'
IF asc%>31 AND asc%<>127
 @display_character
ELSE
 IF LEN(key$)=1
  '
  ' Many Control commands; Return, etc.
  '
  @command_1
  '
  ' Clear keyboard buffer
  '
  LPOKE XBIOS(14,1)+6,0
  '
 ENDIF
ENDIF
'
' *** Mouse Analysis ***
'
' Left mouse button to position cursor.
' Right mouse button for scrolling.
'
IF MOUSEK=1
 @position_cursor
ENDIF
IF MOUSEK=2
 IF MOUSEY<80
  column%=CRSCOL
  PRINT CHR$(27)+"f";
  from%=XBIOS(3)
  post%=XBIOS(3)+1280
 REPEAT
  IF upper%>0
   DEC upper%
   BMOVE from%,post%,30720
   IF block!
    IF upper%+1>=block_start% AND
    upper%+1<=block_end%
     PRINT CHR$(27)+"p";
    ELSE
     PRINT CHR$(27)+"q";
    ENDIF
   ENDIF
   PRINT AT(1,1); CHR$(27)+"l";
   line$(pointer%(upper%+1));
  ENDIF
 UNTIL MOUSEK<>2
 PRINT CHR$(27)+"e";
 PRINT AT(column%,1);
ENDIF
'
IF MOUSEY>320
 column%=CRSCOL
 PRINT CHR$(27)+"f";
 wo%=upper%+26
 from%=XBIOS(3)+1280
 post%=XBIOS(3)
 REPEAT
  IF wo%=<max_lines%
   BMOVE from%,post%,30720
   IF block!
    IF wo%>=block_start% AND wo%<=block_end%
     PRINT CHR$(27)+"p";
    ELSE
     PRINT CHR$(27)+"q";
    ENDIF
   ENDIF
   PRINT AT(1,25); CHR$(27)+"l";
   line$(pointer%(wo%));
  ENDIF
  INC wo%
```

```
   UNTIL MOUSEK<>2
   upper%=wo%-26
   PRINT CHR$(27)+"e";
   PRINT AT(column%,25);
  ENDIF
  '
 ENDIF
 '
LOOP
```

3.6 Basic Editor Functions

We'll start out by writing 'dummy' procedures for the three analysis procedures. Later, we'll come back and complete these procedures.

Another variable is defined by the two command procedures, which is used by the IF THEN statements for branching. In *command_1* this variable is the ASCII code of the pressed key, in *command_2* this is the Scan code of the pressed key.

Since there are time critical functions in *command_2*, a label is placed before the RETURN, which permits jumping over unneeded checks in the procedure.

```
PROCEDURE display_character
RETURN
'
PROCEDURE command1
 command_code%=ASC(Key$)
 ' The IF statements go here.
RETURN
'
PROCEDURE command2
 command_code%=ASC(RIGHT$(Key$))
 ' The IF statements go here.
 end_of_procedure:
RETURN
```

After most IF statements, two kinds of reactions are necessary. First a change on the screen, and secondly, a change of data. Some reactions only occur in one area, for example, with cursor movements.

3.6.1 Moving the Cursor

The string returned by INKEY$ for a cursor keypress is two bytes long. The analysis happens in procedure *command_2*.

All time critical functions in this procedure end with a GOTO end_of_command_2. All of these functions should be at the beginning of the procedure because the cursor up/cursor down keys can cause scrolling.

The cursor position is checked by using the GFA BASIC system variables CRSLIN (Cursor line) and CRSCOL (Cursor Column). CRSLIN contains the screen line where the cursor is located. CRSCOL contains the screen column where the cursor is located. CRSCOL and CRSLIN are effective even if the cursor is invisible. The cursor is moved with VT-52 Escape sequences.

```
IF command_code%=72  ! Cursor up
 IF CRSLIN=1
  @scroll_down
 ELSE
  PRINT CHR$(27)+"A";
 ENDIF
 GOTO command_2_end
ENDIF
```

```
'
'----------
'
IF command_code%=80 ! Cursor down
 IF CRSLIN=25
  @scroll_up
 ELSE
  PRINT CHR$(27)+"B";
 ENDIF
 GOTO command_2_end
ENDIF
'
IF command_code%=77   ! Cursor right
 PRINT CHR$(27)+"C";
ENDIF
'
' ----------
'
IF command_code%=75   ! Cursor left
 PRINT CHR$(27)+"D";
ENDIF
```

3.6.2 Text Input

What is most important in a text editor? Of course, we should be able to type text into it. The next procedure display_character is responsible for just that.

The variable *number_lines* is the number of elements of the array *line$()* which have already been used. The variable *upper%* contains the number of the line which is the upper screen line. Then *upper%+CRSLIN* is then the number of text line, where the cursor is located.

The finding and saving the cursor position can also be done with a VT-52 sequence.

It is important to differentiate between an insert and an overwrite mode. In overwrite mode the new character is simply placed into the string and replaces the old character. In insert mode, a new string has to be put together from several parts and should not get any longer than 80 characters. You must also be aware of whether or not the cursor is in a marked block. The variable *block!* is true if a block has actually been marked.

```
PROCEDURE display_character
 '
 ' Store old cursor position
 char_pos%=CRSLIN
 column%=CRSCOL
 '
 ' Determine text line number which
 ' contains cursor
 '
 where%=upper%+CRSLIN
 '
 ' Turn on Inverse, if cursor is in block.
 '
 IF block!
  IF where%>=block_start% AND
  where%<=block_end%
   PRINT CHR$(27)+"p";
  ELSE
   PRINT CHR$(27)+"q";
  ENDIF
 ENDIF
 '
 ' nr% is the index of the string which
 ' contains the actual line.
 '
 nr%=pointer%(where%)
 '
 ' If the character falls in a line which has
 ' not been used to this point:
 '
 number_lines%=MAX(number_lines%,where%)
 '
 ' Fill line with spaces if no
 ' character is found.
 '
 size%=LEN(line$(nr%))
 IF size%<column%-1
  line$(nr%)=line$(nr%)+SPACE$(column%-size%-1)
 ENDIF
 '
 'If we are in Insert Mode, the line cannot
 'be any longer than 80 characters. The new
 ' line is put together from the string which
 ' was the line preceding the cursor
 ' position, the new character, and the string
 ' which was the following the cursor position.
 '
 IF insertmode!
  IF LEN(line$(nr%))<80
   line$(nr%)=LEFT$(line$(nr%), column%-1)+
   key$+ MID$(line$(nr%),column%)
   PRINT MID$(line$(nr%),column%);
   PRINT AT(column%+1,char_pos%);
  ENDIF
  '
  ' If not in Insert mode, the character simply
  ' replaces the old character.
 ELSE
  MID$(line$(nr%),column%)=key$
  PRINT key$;
 ENDIF
 '
RETURN
```

3.6.3 Delete and Backspace

Now let's look at two of the most important edit functions, the <Delete> and <Backspace> keys. Both return a one character long string through INKEY$, and are evaluated in procedure *command_1*.

The <Backspace> key deletes the character to left of the cursor and pulls the rest of the line onto the deleted character, the cursor also moves one character to the left. If the cursor is already at the left end of a line or if there is no text standing following it, the reactions must be different.

```
IF command_code%=8  ! Backspace Key
 '
 ' Only if cursor is not already at
 ' the left edge of the screen.
 '
 IF CRSCOL>1
  '
  ' Nr% is the index of the actual
  ' line in memory.
  nr%=pointer%(CRSLIN+upper%)
  '
  IF LEN(line$(nr%))>=CRSCOL-1 AND
  LEN(line$(nr%))>=1
   '
   ' Store cursor position
   char_pos%=CRSLIN
   column%=CRSCOL
   '
   ' Rewrite line beginning one character to
   ' the left of the cursor and move cursor
   ' one character to the left.
   '
   PRINT AT(column%-1,char_pos%)
   ;MID$(line$(nr%), column%);" ";
   PRINT AT(column%-1,char_pos%);
   '
   line$(nr%)=LEFT$(line$(nr%),column%-2)+
   MID$(line$(nr%),column%)
  ELSE
   '
   PRINT AT(CRSCOL-1,CRSLIN);
  ENDIF
 ELSE
  PRINT CHR$(7);  ! Error rings bell
 ENDIF
ENDIF
```

When the <Delete> keys is pressed the character under the cursor is removed and the rest of the line moves one character to the left. The cursor position does not change.

```
IF command_code%=127  ! Delete Key
 '
 ' Nr% is the Index of the actual
 ' line in memory.
 nr%=pointer%(CRSLIN+upper%)
 '
 IF LEN(line$(nr%))>=CRSCOL
  '
  ' Store cursor position
  char_pos%=CRSLIN
  column%=CRSCOL
  '
  ' Rebuild line for storage,
  ' with character removed.
  line$(nr%)=LEFT$(line$(nr%),column%-
  1)+MID$(line$(nr%),column%+1)
  '
  ' Display line following cursor position.
  PRINT MID$(line$(nr%),column%);" ";
  PRINT AT(column%,char_pos%);
 ENDIF
ENDIF
```

3.6.4 Return Key

Pressing the <Return> key will cause the cursor to jump to the next line. It the cursor is in the last screen line, then the screen is scrolled.

It is a matter of taste whether pressing the <Return> key should insert a new line. We choose to not implement this feature. If you would like this feature, you can insert a new line in insert mode. By the way, the length of the string returned by INKEY$ is again 1, so this routine is part of the Procedure *command_1*.

```
IF command_code%=13    ! Return Key
 IF CRSLIN=25
  @scroll_up
  PRINT AT(1,25);
 ELSE
  PRINT AT(1,CRSLIN+1);
 ENDIF
ENDIF
```

3.7 Continuing Edit Functions

After all these elementary functions we will go one step further. The editor should be able to do all the essential tasks except communication with the disk and control block operations, which we will discuss in section 3.9.

3.7.1 Adjust Writing Mode

The switching between insert and overwrite mode is done by pressing the <Insert> key. A boolean variable, *insertmode!*, is changed. This variable is checked by other routines. The length of the string returned by INKEY$ is 2.

```
IF command_code%=82    !Insert Key
 insertmode=NOT insertmode
ENDIF
```

3.7.2 Insert Line

The principle of inserting lines was discussed in the section about fast block functions. Remember: All line pointers from the starting location to the location where the text is be inserted (inclusive), are moved up one position. The new pointer then points at the first empty line in line array. Since the first *number_line%* positions are occupied in array *line$()*, this is the line with the index of the raised line number.

The screen is controlled with many VT-52 escape sequences. The line delete function is associated with function key <F3>. Of course this, as well as any other functions, may be changed quite easily. All function keys return only one Scan code and no ASCII code. The string returned by INKEY$ is two characters long.

```
IF command_code%=61   ! F3: Insert Line
 '
 ' Only insert a line if a line is still
 ' available and the cursor is not
 ' following the last line of text.
 '
 IF number_lines%<max_lines% AND
 number_lines%>=upper%+CRSLIN
 '
 ' Insert a blank line with the cursor at
 ' the start of the line.
 '
 PRINT CHR$(27)+"L";
 '
 ' Move pointer up one line from where the
 ' line insertion will take place.
 ' upper%+CRSLIN is the actual line number.
 '
 dest_addr%=base%+(upper%+CRSLIN-1)*4
 source_addr%=dest_addr%+4
 number%=(number_lines%-upper%-CRSLIN+1)*4
 IF number%>0
  BMOVE dest_addr%,source_addr%,number%
 ENDIF
 '
 ' Increment line number and put new
 ' line number at the end of the array.
 '
 INC number_lines%
 pointer%(upper%+CRSLIN)=number_lines%
 '
 ' If a block has been defined, and is
 ' involved in this operation,
 ' adjust accordingly.
 '
 IF block!
  IF upper%+CRSLIN<=block_start%
   INC block_start%
  ENDIF
  IF upper%+CRSLIN<=block_end%
   INC block_end%
  ENDIF
 ENDIF
 '
ELSE
```

```
  PRINT CHR$(7);  ! Ring error bell
 ENDIF
 GOTO command_2_end
ENDIF
```

3.7.3 Delete Line

If a line can be inserted then a line can also be deleted. This is a little bit more complicated because of the limitation that no empty lines can be in front of used lines in string array.

Deleting a line works like this: The line to be deleted is replaced by the last used line of the array *line$()*. Since it is stored twice, it will be deleted at position *line$(number_lines%)*. The pointer, pointing at this line, *number_lines%*, must be found and then changed to the new position.

Then all pointers from the deleted line (plus one) must be moved one line down ('down' meaning: to lower addresses). Finally, the pointer to the line made available by the deletion must be returned to its original position *(pointer%(cnt%)=cnt%)* and the number of lines reduced by one.

One unpleasant factor to consider is looking for the pointer which points at the last line in the array which is *number_lines%*. This is probably at the end of the pointer array and this array is searched from the back to front with a WHILE loop.

The screen management is mainly handled by a VT-52 sequence. Attention must be paid to moving lines up in the lower screen edge if they are part of a defined block.

The function has is called by pressing the <F4> function key. INKEY$ returns a two character string.

```
IF command_code%=62  ! F4: Delete Line
 '
 ' If the cursor is in the text area
 '
 IF upper%+CRSLIN<=number_lines%
  '
  ' Store cursor position and move screen up
  'from cursor
  '
  char_pos%=CRSLIN
  column%=CRSCOL
  '
  ' Delete line which contains cursor and move
  ' remaining lines up one line.
  '
  PRINT CHR$(27)+"M";
  '
  ' Display new line on last screen line, check
  ' if line is part of defined block and treat
  ' accordingly.
  '
  IF block!
   IF upper%+26>=block_start% AND
   upper%+26<=block_end%
    ' Reverse print on
    PRINT CHR$(27)+"p";
   ELSE
    ' Reverse print off
    PRINT CHR$(27)+"q";
   ENDIF
  ENDIF
  PRINT AT(1,25);line$(pointer%(upper%+26));
```

```
'
' Restore cursor at old position
'
PRINT AT(column%,char_pos%);
'
' The preceding lines changed the screen,
' now we must update the lines held in
' memory.
'
' Delete line by moving the last line in
' memory to it's position. Set line to empty
' string.
'
line$(pointer%(upper%+CRSLIN))=
line$(number_lines%)
line$(number_lines%)=""
'
' Find pointer which was pointing to the last
' line and have it point to where the last
' line was moved.
'
cnt%=number_lines%
WHILE pointer%(cnt%)<>number_lines%
 DEC cnt%
WEND
pointer%(cnt%)=pointer%(upper%+CRSLIN)
'
' Move all pointers from the cursor down by
' one array element.
'
dest_addr%=base%+(upper%+CRSLIN)*4
source_addr%=dest_addr%-4
number%=(number_lines%-upper%-CRSLIN)*4
IF number%>0
 BMOVE dest_addr%,source_addr%,number%
ENDIF
```

```
   '
   ' Change pointer of the last line in
   ' it's original position and reduce
   ' the number of lines.
   '
   pointer%(number_lines%)=number_lines%
   DEC number_lines%
   '
  ELSE
   PRINT CHR$(7);  ! Error bell
  ENDIF
  GOTO command_2_end
ENDIF
```

3.7.4 Separate Lines

Often you will encounter the problem that one line of text must be changed into two lines. The line should be broken at the cursor position.

You will notice, that this function is similar to the function which inserts lines. The main difference is that the line which contains the cursor stays the same by inserting a line and the new line is empty. Everything in the line which falls before the cursor position remains the same, and everything following the cursor moves into the new line.

This command is defined as the function key <F8>:

```
IF command_code%=66   ! F8: Split lines
  '
  ' Nr% is the actual screen line
  nr%=upper%+CRSLIN
  '
  ' Only if the cursor is in the text area.
  '
  size%=LEN(line$(pointer%(nr%)))
```

```
IF nr%=<number_lines% AND CRSCOL<=size% AND
number_lines%<max_lines%
 '
 ' The two new line strings:
 '
 part_remaining$=LEFT$(line$(pointer%(nr%)),CR
 SCOL-1)
 cutoff$=MID$(line$(pointer%(nr%)),CRSCOL)
 '
 ' Delete actual line beginning with cursor
 ' position and move screen down one line in
 ' order to write text to the next line.
 '
 ' VT-52 Escape to clear from cursor
 ' to the end of the line:
 '
 PRINT CHR$(27)+"K";
 IF CRSLIN<25
  '
  ' Insert a blank line with the
  ' cursor at the start of the line.
  '
  PRINT AT(1,CRSLIN+1);CHR$(27)+"L";
  PRINT AT(1,CRSLIN);cutoff$;
 ENDIF
 '
 ' Increment line numbers and move all line
 ' pointers up starting at the inserted line.
 '
 INC number_lines%
 dest_addr%=base%+nr%*4
 source_addr%=dest_addr%+4
 number%=(number_lines%-nr%-1)*4
 IF number%>0
  BMOVE dest_addr%,source_addr%,number%
 ENDIF
 '
 ' Place the two new lines in memory.
 '
 pointer%(nr%+1)=number_lines%
```

```
   line$(pointer%(nr%))=part_remaining$
   line$(pointer%(nr%+1))=cutoff$
   '
   PRINT AT(1,CRSLIN);
  ELSE
   PRINT CHR$(7);  ! Error bell
  ENDIF
ENDIF
```

3.7.5 Combining Lines

If lines can be separated, then there should also be a way of combining two lines. The following function moves the line following the cursor into the line containing the cursor, if the new line is not longer than 80 characters.

Naturally, the total length of the text will be shortened by one line. For that reason, this function resembles the delete line function. This command is called by pressing the <F7> function key.

```
IF command_code%=65  ! F7: Join lines
 '
 ' Only if another line follows
 IF upper%+CRSLIN<number_lines%
  '
  ' nr% is the actual line number.
  '
  nr%=upper%+CRSLIN
  '
  ' Only if the two lines will not
  ' exceed 80 characters.
  '
```

```
IF LEN(line$(pointer%(nr%)))+
LEN(line$(pointer%(nr%+1)))<81
 '
 ' Put new line together
 ' and display on screen.
 '
 line$(pointer%(nr%))=line$(pointer%(nr%))+li
 ne$(pointer%(nr%+1))
 PRINT AT(1,CRSLIN);line$(pointer%(nr%));
 '
 '
 ' Store cursor position and move cursor down
 ' one line.
 ' Move screen up one line.
 '
 char_pos%=CRSLIN
 column%=CRSCOL
 PRINT AT(column%,char_pos%+1);
 '
 ' Delete line with cursor and move
 ' remaining lines up one line.
 '
 PRINT CHR$(27)+"M";
 '
 IF block!
  IF cnt%>=block_start% AND cnt%<=block_end%
   ' Reverse text on
   PRINT CHR$(27)+"p";
  ELSE
   ' Reverse text off
   PRINT CHR$(27)+"q";
  ENDIF
 ENDIF
 PRINT AT(1,25);line$(pointer%(upper%+26));
 PRINT AT(column%,char_pos%);
 '
 ' Delete line number nr%+1
 '
 ' Delete line by moving last line
 ' to its position.
 '
 line$(pointer%(nr%+1))=line$(number_lines%)
 '
 ' Clear last line
 '
 line$(number_lines%)=""
 '
 ' Look for pointer to last line and change
 ' it to point to the where the line was
 ' moved.
 '
 cnt%=number_lines%
 WHILE pointer%(cnt%)<>number_lines%
  DEC cnt%
 WEND
 pointer%(cnt%)=pointer%(nr%+1)
 '
 ' Change pointer one element down from
 ' the line to be deleted.
 '
 dest_addr%=base%+(nr%+1)*4
 source_addr%=dest_addr%-4
 number%=(number_lines%-nr%-1)*4
 IF number%>0
  BMOVE dest_addr%,source_addr%,number%
 ENDIF
 '
 ' Return last pointer to original condition,
 ' and reduce line numbers.
 '
 pointer%(number_lines%)=number_lines%
 DEC number_lines%
 '
ELSE
```

```
    ALERT 1,"The Lines are too long|(New line
    over 80 Characters)",1," Return ",a%
   ENDIF
  ELSE
   PRINT CHR$(7);   ! Error bell
  ENDIF
ENDIF
```

3.7.6 Scrolling

The main program and the various routines use the next two routines which control screen scrolling.

The functions of these two routines have been explained in a previous section. The Procedure scroll_up moves the screen up.

```
PROCEDURE scroll_up
  '
  ' If the last line has not been reached.
  IF upper%+26=<max_lines%
    '
    ' Increment line counter, store cursor
    ' position and turn off cursor.
    '
    INC upper%
    column%=CRSCOL
    '
    ' VT-52 Escape sequence to hide the cursor
    '
    PRINT CHR$(27)+"f";
  '
  ' Move the screen
  '
  BMOVE XBIOS(3)+1280,XBIOS(3),30720
  '
  ' Use inverse text if new line is part
  ' of a defined block.
  '
  IF block!
   IF upper%+25>=block_start% AND
   upper%+25<=block_end%
    ' Reverse on
    PRINT CHR$(27)+"p";
   ELSE
    ' Reverse off
    PRINT CHR$(27)+"q";
   ENDIF
  ENDIF
  '
  ' Turn cursor on, delete line
  ' and display new line.
  '
  PRINT AT(1,25); CHR$(27)+"e"; CHR$(27)+"l";
  line$(pointer%(upper%+25));
  PRINT AT(column%,CRSLIN);
  '
 ELSE
  PRINT CHR$(7); ! Error bell
 ENDIF
RETURN
```

In order to scroll towards the start of the text, the screen must be scrolled down.

```
PROCEDURE scroll_down
 '
 ' If the first line has not been reached:
 '
 IF upper%>0
  '
  ' reduce line counter, store cursor position,
  ' and turn off cursor.
  '
  DEC upper%
  column%=CRSCOL
  '
  ' VT-52 escape sequence to hide the cursor.
  '
  PRINT CHR$(27)+"f";
  '
  ' Move the screen
  '
  BMOVE XBIOS(3),XBIOS(3)+1280,30720
  '
  ' If line is part of a defined block, use
  ' reverse text mode.
  '
  IF block!
   IF upper%+1>=block_start% AND
   upper%+1<=block_end%
    ' Reverse on
    PRINT CHR$(27)+"p";
   ELSE
    ' Reverse off
    PRINT CHR$(27)+"q";
   ENDIF
  ENDIF
```

```
   '
   ' Turn of cursor, delete line,
   ' and display new line.
   '
   PRINT AT(1,1); CHR$(27)+"e"; CHR$(27)+"l";
   line$(pointer%(upper%+1));
   PRINT AT(column%,CRSLIN);
   '
  ELSE
   PRINT CHR$(7);   ! Error bell
  ENDIF
RETURN
```

Here is a faster scrolling routine that is part of the main program loop to be used when the mouse is selected.

```
IF MOUSEK=2
 IF MOUSEY<80
  column%=CRSCOL
  PRINT CHR$(27)+"f";
  from%=XBIOS(3)
  post%=XBIOS(3)+1280
  REPEAT
   IF upper%>0
    DEC upper%
    BMOVE from%,post%,30720
    IF block!
     IF upper%+1>=block_start% AND
     upper%+1<=block_end%
      PRINT CHR$(27)+"p";
     ELSE
      PRINT CHR$(27)+"q";
     ENDIF
    ENDIF
    PRINT AT(1,1); CHR$(27)+"l";
    line$(pointer%(upper%+1));
   ENDIF
  UNTIL MOUSEK<>2
  PRINT CHR$(27)+"e";
  PRINT AT(column%,1);
 ENDIF
 '
 IF MOUSEY>320
  column%=CRSCOL
  PRINT CHR$(27)+"f";
  where%=upper%+26
  from%=XBIOS(3)+1280
  post%=XBIOS(3)
  REPEAT
   IF where%=<max_lines%
    BMOVE from%,post%,30720
    IF block!
     IF where%>=block_start% AND
     where%<=block_end%
      PRINT CHR$(27)+"p";
     ELSE
      PRINT CHR$(27)+"q";
     ENDIF
    ENDIF
    PRINT AT(1,25); CHR$(27)+"l";
    line$(pointer%(where%));
   ENDIF
   INC where%
  UNTIL MOUSEK<>2
  upper%=where%-26
  PRINT CHR$(27)+"e";
  PRINT AT(column%,25);
 ENDIF
 '
ENDIF
```

3.7.7 Moving by Pages

Normal scrolling is a line by line process. Often you'll need to move faster than this permits. For larger jumps, an option of moving a page at a time is provided. This option is defined as the function keys <F9> (paging up) and <F10> (paging down).

The functions of the routines are very simple. The value of the variable *upper%* is changed by 20 and the text is displayed beginning with this line by the procedure output_from_line() is used.

Special attention must be used to insure that the start and the end of the text is handled properly and that any defined blocks are marked correctly.

```
IF command_code%=67   ! F9: Page up
  IF upper%>0
    '
    ' Store cursor position
    char_pos%=CRSLIN
    column%=CRSCOL
    '
    ' If no whole page precedes the page on which
    ' the cursor is positioned, move to beginning
    ' of text.
    '
    IF upper%<20
      upper%=0
      @output_from_line(1)
    ELSE
      SUB upper%,20
      @output_from_line(upper%+1)
    ENDIF
    PRINT AT(column%,char_pos%);
    '
  ELSE
    ' Error bell, if no other page.
    PRINT AT(1,1);CHR$(7);
  ENDIF
  GOTO command_2_end
ENDIF
'
' ----------
'
```

```
IF command_code%=68  ! F10: Page down
 IF upper%<max_lines%-25
  '
  ' Store Cursor position
  char_pos%=CRSLIN
  column%=CRSCOL
  '
  ' If no whole page follows the cursor
  ' position, display to end of text.
  '
  IF upper%>max_lines%-48
   upper%=max_lines%-25
   @output_from_line(upper%+1)
  ELSE
   ADD upper%,20
   @output_from_line(upper%+1)
  ENDIF
  PRINT AT(column%,char_pos%);
  '
 ELSE
  ' Error bell if no page exists.
  PRINT AT(1,1);CHR$(7);
 ENDIF
 GOTO command_2_end
ENDIF
```

All lines are displayed with PRINT commands, without using a semicolon behind the PRINT statement. Only the last screen line uses a PRINT statement with a semicolon.

```
PROCEDURE output_from_line(nr%)
 CLS
 '
```

```
' If text is here:
IF number_lines%>0
 '
 ' Display line either to line number nr%+24
 ' or to the end of the text.
 '
 til%=MIN(nr%+24,number_lines%)
 '
 ' In case nr% is already at the last line:
 '
 IF nr%=number_lines%
  IF block!
   IF nr%>=block_start% AND nr%<=block_end%
    ' Reverse on
    PRINT CHR$(27)+"p";
   ELSE
    ' Reverse off
    PRINT CHR$(27)+"q";
   ENDIF
  ENDIF
  PRINT line$(pointer%(nr%))
 ENDIF
 '
 ' If nr% is somewhere near the beginning of
 ' the text, display lines up to til%-1
 '
 IF nr%<number_lines%
  '
  ' Display all lines from nr% to til%-1
  ' with linefeed.
  '
  IF til%>nr%
   FOR cnt%=nr% TO til%-1
    '
  ' In case of defined block, display in
  ' inverse text.
  '
  IF block!
   IF cnt%>=block_start% AND
   cnt%<=block_end%
    ' Reverse on
    PRINT CHR$(27)+"p";
   ELSE
    ' Reverse off
    PRINT CHR$(27)+"q";
   ENDIF
  ENDIF
  '
  ' Output with linefeed:
  '
  PRINT line$(pointer%(cnt%))
 NEXT cnt%
ENDIF
'
' Output line til% without linefeed.
' Automatic line overflow
' is turned off, even for lines with a
' length of over 80 characters in the last
' line to prevent scrolling.
'
' In case of a defined block,
' use inverse mode.
'
IF block!
 IF til%>=block_start% AND til%<=block_end%
  ' Reverse on
  PRINT CHR$(27)+"p";
 ELSE
  ' Reverse off
  PRINT CHR$(27)+"q";
 ENDIF
ENDIF
```

```
   '
   ' Last line without line feed.
   '
   PRINT line$(pointer%(til%));
   '
  ENDIF
 ENDIF
 '
RETURN
```

3.7.8 Search a String within the Text

Searching for a string within the text segment should present the same possibilities as are available in the GFA BASIC Editor. Pressing the respective function keys enables input of a search string.

Pressing <Control> <F> repeats the search for the specified string. Since this function is simpler, we'll examine it first.

Stated simply, the algorithm is: If a search string exists, begin searching with the line which follows the cursor.

```
IF command_code%=6  ! Control+F: Search for text
 IF search_string$<>""
  @search_from(upper%+CRSLIN+1)
 ENDIF
ENDIF
```

Since <Control> + <F> produces a one-character-Inkey$-String, these few characters go into procedure command1.

Pressing the <F5> function key makes it possible to specify and search for a string. A line on the upper portion of the screen is displayed for entering the string to search for. The upper line of the screen is stored with a GET command in order to restore it later and the prompt line is removed. After the screen is restored, the search procedure is called.

```
IF command_code%=63  ! F5: Find
 '
 ' VT-52 Escape sequence to hide cursor
 PRINT CHR$(27)+"f";
 '
 ' store upper screen area and
 ' cursor position
 '
 GET 0,0,639,20,top_scrn$
 char_pos%=CRSLIN
 column%=CRSCOL
 '
 ' Clear upper line and get search string
 '
 PRINT AT(1,1);SPACE$(80);
 LINE 0,18,639,18
 LINE 0,19,639,19
 PRINT AT(1,1);"Search String: "
 PRINT AT(13,1);
 FORM INPUT 60 AS search_string$
 '
 ' Restore screen and call search routine
 '
 PUT 0,0,top_scrn$
 '
 ' VT-52 Escape sequence to show cursor
 '
 PRINT CHR$(27)+"e";
 IF search_string$<>""
  @search_from(upper%+char_pos%+1)
 ELSE
  PRINT AT(column%,char_pos%);
 ENDIF
ENDIF
```

In both routines a procedure called *search_from* was used, which passes as a parameter the text line number containing the starting point for the search.

First, a check is made to be sure the line is still in the text area. Then a loop using the INSTR command is used to search the strings in the text array. If the search string is found, the search loop stops and the variable *found!* is set to true (-1). If the search was successful, the number of the line where the search string was found is stored in *cnt%*.

If the search was successful, the text is displayed on the screen beginning with this line. If it was not successful, the end of the text segment will be displayed on the screen. The line with the MAX command is necessary with text lengths of less than 25 lines, when the search is unsuccessful.

```
PROCEDURE search_from(t_line%)
 found!=FALSE
 '
 ' Only if the cursor is in the text area.
 IF t_line%<=number_lines%
  '
  ' Search all lines until the
  ' search string is found.
  '
  FOR cnt%=t_line% TO number_lines%
   IF INSTR(line$(pointer%(cnt%)),
   search_string$)<>0
    found!=TRUE
   ENDIF
   EXIT IF found!
  NEXT cnt%
 ENDIF
 '
 ' Display text beginning with found location.
 '
 IF found!
  upper%=MAX(0,cnt%-1)
  @output_from_line(cnt%)
  PRINT AT(1,1);
 ELSE
  upper%=MAX(0,number_lines%-25)
  @output_from_line(upper%+1)
  ALERT 1,"Search String not found.",1," Return
  ",a%
 ENDIF
RETURN
```

By the way, this part of the program is rather fast.

3.7.9 Replace Text

This routine is similar to the one used for searching. A separate routine for repeating the replacement of a specified string is called by pressing <Control> <R>. This is par of the procedure *command_1*.

```
IF command_code%=18
' Control+R: Search and replace
 IF search_string$<>""
  @replace_from(upper%+CRSLIN)
 ENDIF
ENDIF
```

When function key <F6> is pressed, the search and replacement strings may be specified.

```
IF command_code%=64  ! F6: Replace
 '
 ' VT-52 Escape code to hide cursor
 '
 PRINT CHR$(27)+"f";
 '
 ' Save upper screen area and cursor position
 '
 GET 0,0,639,36,top_scrn$
 char_pos%=CRSLIN
 column%=CRSCOL
 '
 ' Delete upper two screen lines and
 ' use them to get the search and replace
 ' strings.
 '
 PRINT AT(1,1);SPACE$(80);
 PRINT AT(1,2);SPACE$(80);
 LINE 0,34,639,34
 LINE 0,35,639,35
 PRINT AT(1,1);"Search for: "
 PRINT AT(1,2);"Replace with: "
 PRINT AT(15,1);
 FORM INPUT 60 AS search_string$
 PRINT AT(15,2);
 FORM INPUT 60 AS replace_string$
 '
 ' Restore upper lines of screen.
 '
 PUT 0,0,top_scrn$
 '
 ' VT-52 Escape code to turn on cursor
 '
 PRINT CHR$(27)+"e";
 '
 ' Start search and replace
 IF search_string$<>""
  @replace_from(upper%+char_pos%)
 ENDIF
ENDIF
```

Here's the subroutine which will do the actual search and replace operation. It should look somewhat familiar to you.

```
PROCEDURE replace_from(t_line%)
 found!=FALSE
 '
 ' Only if search is conducted in text area.
 IF t_line%<=number_lines%
  '
  ' Search until found, then exit.
  '
  FOR cnt%=t_line% TO number_lines%
   where%=INSTR(line$(pointer%(cnt%)),
   search_string$)
   IF where%<>0
    found!=TRUE
   ENDIF
   EXIT IF found!
  NEXT cnt%
  '
 ENDIF
 found_at%=cnt%
 '
 IF found!
  '
  ' If string gets too long by replacement:
  '
  IF LEN(line$(pointer%(cnt%)))+
  LEN(replace_string$)-LEN(search_string$)>80
   ALERT 1,"The replacement line is|going to be
   too|long!",1," Return ",a%
  ELSE
   '
   ' Nr% is the index of the line where the
   ' replacement takes place.
   '
   nr%=pointer%(cnt%)
   '
   ' Do replacement
   '
```

```
    line$(nr%)=LEFT$(line$(nr%),where%-1)+
    replace_string$+MID$(line$(nr%),
    where%+LEN(search_string$))
   ENDIF
   '
   ' Display text from point of discovery.
   '
   upper%=cnt%-1
   @output_from_line(upper%+1)
   PRINT AT(1,1);
 ELSE
   '
   ' In case search string was not found:
   '
   upper%=MAX(0,number_lines%-25)
   @output_from_line(upper%+1)
   ALERT 1,"Search string was not found.",1,"
Return ",a%
   '
 ENDIF
 '
RETURN
```

3.7.10 Delete Text

Here are two important, but not very difficult functions. First, deleting all of the actual text. A few variables are changed, all used lines are deleted and the pointers are reset to their original values. The screen is then restored with the CLS command.

This function is called by pressing the <Undo> key, and the string returned by INKEY$ has a length of two characters; therefore this listing is part of the procedure *command_2*.

```
IF command_code%=97
' Undo Key for deleting all of text
 IF number_lines%>0
  ALERT 3," |Do you really want|to delete all
  of|the text?",1," Yes | No ",a%
  IF a%=1
   '
   ' Delete all lines and restore pointer
   ' to it's original position.
   '
   FOR cnt%=1 TO number_lines%
    pointer%(cnt%)=cnt%
    line$(cnt%)=""
   NEXT cnt%
   '
   ' Reset variables
   '
   upper%=0
   number_lines%=0
   block!=FALSE
   block_start%=0
   block_end%=0
   CLS
   '
  ENDIF
 ENDIF
ENDIF
```

3.7.11 End of Program

The other important, but equally simple function is 'Quit'. We used the <Esc> key, because of the association of the words 'escape' and 'quit'. Since there is a chance of pressing this key accidentally, the <Control> key must be pressed at the same time. The code from the generated by the <Esc> key does not change by pressing the <Control> key, this must be checked by using BIOS(11,-1) Also, a safety check should be used, as in any function that can lead to a loss of data.

The string returned by INKEY$ is one character long, consequently this listing is part of the procedure *command_1*.

```
IF command_code%=27
' Escape key plus Control = End Program
 '
 ' Control key must also be pressed.
 '
 IF BIOS(11,-1)=4
  ALERT 3," |End Program and|Lose any
  unsaved|work?",1," Yes | No ",a%
  IF a%=1
   ' VT-52 Escape sequence Hide Cursor
   PRINT CHR$(27)+"f";
   ' VT-52 Escape Sequence Inverse text mode
   PRINT CHR$(27)+"q";
   EDIT
  ENDIF
 ENDIF
ENDIF
```

3.8. Communication with the Outside World

Now we are ready to discuss the function which permit the computer to communicate with external devices, such as the disk drives and printer.

3.8.1 Loading Text

Loading text appears to be very simple:

```
number_lines%=0
REPEAT
 INC Number_lines%
 LINE INPUT #1,line$(number_lines%)
UNTIL EOF(#1)
```

Unfortunately, this algorithm is very slow for loading large files. There is a much faster method. This method has the disadvantage of being much more work for the programmer. Remember the fast loading method for arrays used in the 3D Graphic program? A method that is somewhat similar is also used for strings, z$=INPUT$(n%,#k%), where exactly n% bytes is read from data channel k% and stored in z$ string.

Each line is ended with two bytes, a 'CR', (ASCII Code 13) for Carriage Return and 'LF' (ASCII Code 10) for Line Feed. Both terms originated with typewriters.

We must search for one of these characters (in our program, the LF), and divide the data read into bytes, which are available as a string in the line array.

The algorithm is as follows. An INPUT$ command will read *n* bytes. We defined *n* as 256. We will calculate from the size of the file how many 256 byte segments can be read in the loop. The remaining bytes may then be read with a command.

The strings are searched with the INSTR command for the LF codes. When one is found, everything before that character is packed into a line of the line array. The rest is stored for the next line. The CR and LF codes are not stored as part of the array.

Since lines in the editor can not be any longer than 80 characters, longer lines are divided into several lines by the procedure *divide_long_lines*.

It is inconvenient for the user to have to adjust the fileselector for the path name every time it's called. As the programmer, you should write the fileselector routines so that the path name can be stored. In our text editor, the procedure *memorize_path* stores the path. Also, you should check for a filename being selected, as well as the selection of the 'Cancel'.

```
IF command_code%=59  ! F1: Load Text
 IF number_lines%>0
  ALERT 3,"Warning!| |Any text in the
  Editor|will be deleted!",1," Load | Abort
  ",a%
 ELSE
  a%=1
 ENDIF
```

```
IF a%=1
  '
  ' Delete all lines and return pointer
  ' to its original position.
  '
  FOR cnt%=1 TO number_lines%
   pointer%(cnt%)=cnt%
   line$(cnt%)=""
  NEXT cnt%
  '
  ' Reset Variables
  '
  upper%=0
  number_lines%=0
  block!=FALSE
  block_start%=0
  block_end%=0
  '
  '
  ' Hide cursor
  '
  PRINT CHR$(27)+"f"; !
  '
  ' Fileselector with Title
  CLS
  BOX 157,29,482,53
  BOX 159,31,480,51
  TEXT 279,47,"Load Text"
  FILESELECT path$,"",filename$
  @store_path
  '
  ' If OK box is selected with a filename
  ' or the Cancel button is not selected:
  '
  IF NOT (RIGHT$(filename$)="\" OR
  filename$="")
```

```
'
' Open file, read 256 bytes and search
' for the string end character (LF)
' then concatenate strings.
'
OPEN "I",#1,filename$
'
' Define some variables and initialize.
'
size%=LOF(#1)
passes%=INT(size%/256)
rest%=size% MOD 256
t_string$=""
number%=0
cnt%=0
'
' If at least 256 bytes are available:
'
IF passes%>0
 '
 ' Read all 256 bytes
 '
 REPEAT
  INC cnt%
  '
  ' Now the bytes are read
  '
  temp$=INPUT$(256,#1)
  t_string$=t_string$+temp$
  '
  ' Search for end of string character (LF)
  '
  where%=INSTR(t_string$,CHR$(10))
  '
  ' Concatenate strings until there
  ' is no end string character.
  ' Then read any additional bytes.
  '
  WHILE where%>0 AND number%<max_lines%
   INC number%
   '
   ' Characters before CR character.
   '
   line$(number%)=LEFT$(t_string$,where%-2)
   IF LEN(line$(number%))>80
    @divide_long_lines
   ENDIF
   '
   ' Characters after the LF
   '
   t_string$=MID$(t_string$,where%+1)
   where%=INSTR(t_string$,CHR$(10))
  WEND
 UNTIL cnt%=passes% OR number%>=max_lines%
ENDIF
'
' Now read the rest of the bytes which did
' not fit into a 256 byte segment.
'
IF rest%>0 AND number%<max_lines%
 rest$=INPUT$(rest%,#1)
 t_string$=t_string$+rest$
 where%=INSTR(t_string$,CHR$(10))
 WHILE where%>0 AND number%<max_lines%
  INC number%
  line$(number%)=LEFT$(t_string$,where%-2)
  IF LEN(line$(number%))>80
   @divide_long_lines
  ENDIF
  t_string$=MID$(t_string$,where%+1)
  where%=INSTR(t_string$,CHR$(10))
 WEND
ENDIF
'
' If the data ends with a character string
' which does not end with a LF, it can be
' read here.
'
IF LEN(t_string$)>0 AND number%<max_lines%
 INC number%
```

```
    line$(number%)=t_string$
    IF LEN(line$(number%))>80
     @divide_long_lines
    ENDIF
   ENDIF
   '
   IF number%=max_lines%
    ALERT 1,"The selected file was too
    large.|File not loaded.",1," Return ",a%
   ENDIF
   CLOSE #1
   '
   ' Remove older parts of text.
   '
   IF number%<number_lines%
    FOR j%=number%+1 TO number_lines%
     line$(j%)=""
     pointer%(j%)=j%
    NEXT j%
   ENDIF
   number_lines%=number%
  ENDIF
  '
  upper%=0
  block!=FALSE
  block_start%=0
  block_end%=0
  '
 ENDIF
 '
 CLS
 '
 ' VT-52 Escape for turning on cursor
 '
 PRINT CHR$(27)+"e"
 @output_from_line(1)
 PRINT AT(1,1);
 '
ENDIF
```

Here is the support procedure, *store_path.*

```
PROCEDURE store_path
  '
  ' Routine to store the path name
  ' for the Fileselector
  '
  IF filename$<>""
   path$=filename$
   IF RIGHT$(path$)<>"\"
    FOR cnt%=LEN(filename$) DOWNTO 1
     path$=LEFT$(path$,LEN(path$)-1)
     EXIT IF RIGHT$(path$)="\"
    NEXT cnt%
    path$=path$+"*.*"
   ENDIF
  ENDIF
RETURN
```

Here's the routine for splitting up lines with more than 80 characters.

```
PROCEDURE divide_long_lines
  WHILE LEN(line$(number%))>80 AND
  number%<max_lines%
    z$=line$(number%)
    line$(number%)=LEFT$(z$,80)
    INC number%
    line$(number%)=MID$(z$,81)
  WEND
RETURN
```

3.8.2 Storing Text

After this rather difficult loading routine, we are ready for the much easier storing routine. The data is simply sent to the disk in a loop line by line.

```
IF command_code%=60   ! F2: Store Text
  IF number_lines%>0
   char_pos%=CRSLIN
```

```
 column%=CRSCOL
 '
 ' Hide cursor
 '
 PRINT CHR$(27)+"f";
 '
 ' Fileselector with title
 '
 CLS
 BOX 157,29,482,53
 BOX 159,31,480,51
 TEXT 263,47,"Save Text"
 FILESELECT path$,"",filename$
 @store_path
 '
 ' If not OK without filename or Cancel Button
 IF NOT (RIGHT$(filename$)="\" OR
 filename$="")
  '
  ' Open file for output. If file already
  ' exists delete the old file and write
  ' data line by line.
  '
  OPEN "O",#1,filename$
  FOR cnt%=1 TO number_lines%
   PRINT #1,line$(pointer%(cnt%))
  NEXT cnt%
  CLOSE #1
  '
 ENDIF
 '
 CLS
 ' Show cursor
 PRINT CHR$(27)+"e"
 @output_from_line(upper%+1)
 PRINT AT(column%,char_pos%);
 '
ELSE
```

```
  ALERT 1," |There is no text to store!",1,"
  Return ",a%
 ENDIF
ENDIF
```

3.8.3 Printing Text

Printing is just as easy as saving to the disk. We'll just use a loop with LPRINT. To print any special characters use the printer program from your GFA BASIC disk, which takes care of any adjustments.

If the printer is not ready when data is sent to it, the computer 'hangs'. In the simplest case, this problem can be solved by just turning on the printer. But sometimes this is not possible. For instance, no paper is in the printer and you want to continue editing your work.

The operating system of ST waits for about 30 seconds for the printer. This is not necessary, since the operating system has a built-in routine which can immediately notify you if a printer is ready. This routine, GEMDOS (17), is used in the following listing.

Since a printout routine is also necessary to print a block, the actual printing routine was defined as a procedure, containing the start and end line of the printout as parameters.

The command to print is <Control> <P>, and the procedure has to be imbedded in *command1*.

```
IF command_code%=16    !Control + P    Print text
 IF number_lines%>0
  ALERT 2,"Print Text?",1," Yes | No ",a%
  IF a%=1
   @to_printer(1,number_lines%)
  ENDIF
 ENDIF
ENDIF
```

The actual printing procedure is:

```
PROCEDURE to_printer(from%,til%)
 IF GEMDOS(17)
  FOR cnt%=from% TO til%
   LPRINT line$(pointer%(cnt%))
  NEXT cnt%
 ELSE
  ALERT 1,"Printer is not ready.",1," Return
  ",a%
 ENDIF
RETURN
```

3.9. Block Operations

Now we are getting to the most difficult part of the editor, the block operations. We'll start with two easy functions.

3.9.1 Marking a Block and Clearing a Mark

A block can be described with three variables. The variable *block!* is a boolean variable and is true (-1) when a block has been defined. Blockbegin% contains the number of the line where the block begins and blockend% contains the line-number of the last blockline.

The marking of a block can be completed by using just these three variables. Other program parts must deal with the correct management of a block.

All Block operations can be accessed by using either a function-key, or a shifted-function key. The marking of the start of a block is handled by <Shift> + <F1>. The end of a block is marked by pressing <Shift> + <F2>. The marked block may be canceled by pressing <Shift> + <F10>.

First let's look at marking the beginning of a block. It is not enough to simply write this:

```
blockbegin%=uppermost%+CRSLIN
```

even so, this is the most important command to mark the beginning of the block. A few aspects are to be considered. *block_start* cannot be shown on the screen without finding the *block_end* marking. Also, the cursor must be in the text segment and not following it. The screen must also be redrawn with the block shown in reverse mode text. Scrolling routines and the procedure *output_from_line()* are responsible for displaying the text correctly.

```
IF command_code%=84
' Shift+F1: Mark beginning of block
  '
  ' If cursor is in text area
  IF upper%+CRSLIN<=number_lines%
   previous!=block!
   block_start%=upper%+CRSLIN
   '
   'If the newly marked block falls before
   ' block_end:
   '
   IF block_end%>=block_start%
    block!=TRUE
   ELSE
    block!=FALSE
   ENDIF
  ENDIF
  '
  ' Display new text
  '
  char_pos%=CRSLIN
  column%=CRSCOL
  @output_from_line(upper%+1)
  PRINT AT(column%,char_pos%);
ENDIF
```

The marking of the end of the block with *block_end* is similar. In this editor, the line with the cursor is included in the block. The GFA BASIC editor and some other text editors end the defined block with the line before the cursor. If you like that better, you can change it by using: blockend%=upper%+CRSLIN-1.

```
IF command_code%=85
 ' Shift+F2: Mark end of block
 '
 ' If cursor is in text:
 IF upper%+CRSLIN<=number_lines%
  previous!=block!
  block_end%=upper%+CRSLIN
  '
  ' Only if the end of the block is after
  ' the beginning of the block.
  '
  IF block_end%>=block_start%
   block!=TRUE
  ELSE
   block!=FALSE
  ENDIF
 ENDIF
 '
 ' Display text:
 '
 char_pos%=CRSLIN
 column%=CRSCOL
 @output_from_line(upper%+1)
 PRINT AT(column%,char_pos%);
ENDIF
```

Canceling the marked block is even easier. Just set *block!=FALSE* and the defined block marks are cleared. If you’d like, you can even make the output dependent on the availability of blocklines on the screen. On the other hand, refreshing the screen is a good way to inform that a command has been completed.

```
IF command_code%=93
' Shift+F10: Cancel block marks
 IF block!
  block!=FALSE
  PRINT CHR$(27)+"q";
  IF upper%+25>=block_start% AND
  upper%<=block_end%
   @output_from_line(upper%+1)
  ENDIF
  block_start%=0
  block_end%=0
 ENDIF
ENDIF
```

3.9.2 Delete Block

The Delete Block function works like this: First all lines of the block in memory are deleted by moving the last lines of the array *line$()*. Since they will then appear twice, they are deleted at the end of the memory area. Also, the pointers are reset.

The same problems occur as were encountered in the delete line function. The pointer to the last line of the memory block, *line$(number_lines%)* must be found. This procedure has to be done for each line of the block. For large amounts of text and a large block, this will probably cause delays. The true reason for this awkward search is that we don't want any unused lines in memory placed in front of a used line.

Think about possible algorithms for doing a block delete. Perhaps you can find a more elegant and faster method.

After the FOR loop the pointers which follow the end of the block are moved. This deletes all pointers of block lines. Then the text following the block is moved to the front.

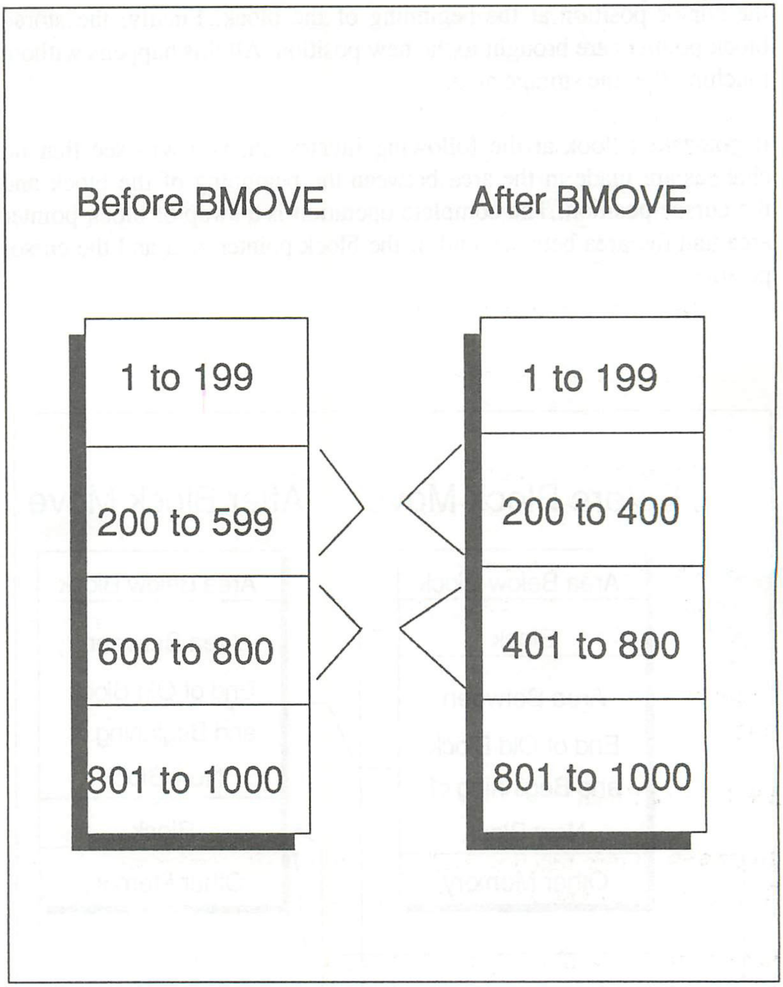
Then all that remains to be done is to set the pointers at the end of pointer array to their original condition. The line number must be reduced by changing the variable number_lines%. The screen must also be redrawn, and the work is completed.

```
IF command_code%=91 ! Shift+F8: Delete Block
 IF block!
  '
  blocksize%=block_end%-block_start%+1
  replace%=block_start%
  through%=number_lines%
  '
  ' Lines of the block at the end of the array:
  '
  FOR cnt%=1 TO blocksize%
   line$(pointer%(replace%))=line$(through%)
   line$(through%)=""
   '
   ' Find the pointer which was pointing
   ' at the line with the number through% and
   ' swap it with the pointer to the
   ' line pointer%(replace%)
   '
   j%=number_lines%
   WHILE pointer%(j%)<>through%
    DEC j%
   WEND
   SWAP pointer%(replace%),pointer%(j%)
   INC replace%
   DEC through%
  NEXT cnt%
  '
  ' Move the pointer array by the length of the
  block, overwrite the blockline pointers.
  '
  dest_addr%=base%+block_end%*4
  source_addr%=base%+(block_start%-1)*4
  number%=(number_lines%-block_end%)*4
  IF number%>0
   BMOVE dest_addr%,source_addr%,number%
  ENDIF
  '
  ' Reduce the number of lines,
  ' and return the pointer of the cleared lines
  ' to original state.
  '
  SUB number_lines%,blocksize%
  FOR cnt%=number_lines%+1 TO
  number_lines%+blocksize%
   pointer%(cnt%)=cnt%
  NEXT cnt%
  '
  ' Set block markings
  '
  upper%=block_start%
  block!=FALSE
  '
  ' Turn off reverse text mode
  ' and redraw screen:
  '
  PRINT CHR$(27)+"q";
  @output_from_line(upper%+1)
  PRINT AT(1,1);
  '
 ENDIF
ENDIF
```

3.9.3 Move Block

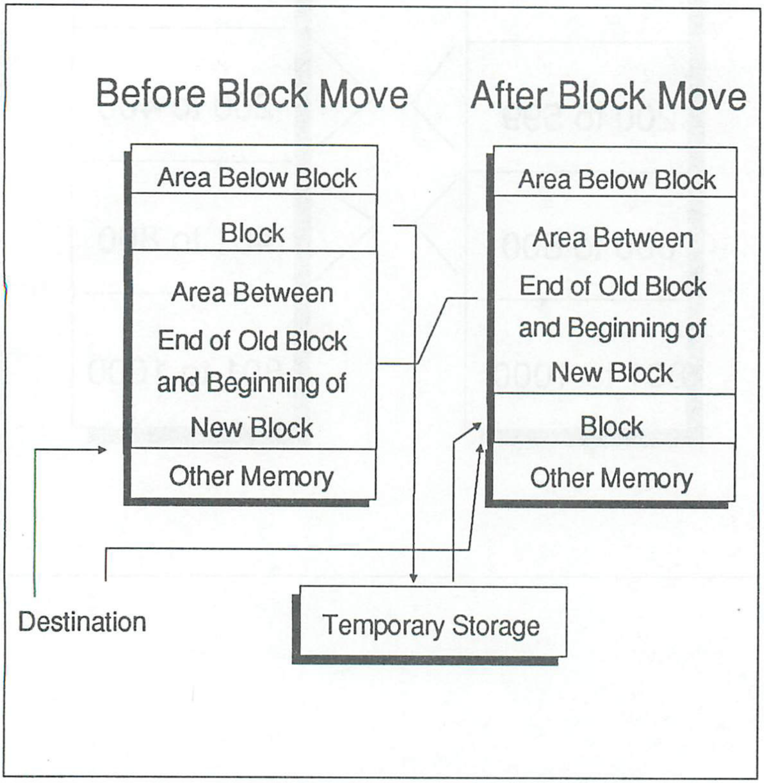
The Move Block function will use the BMOVE command of GFA BASIC, it's very fast, and doesn't need any loops.

Before moving a block, you must check that the destination of the block is in front or behind its old position. Let's take a look at the case, where the destination follows the block. Since the destination is the same as the cursor position, we can also say that in this case the block is before the cursor line.


Before BMOVE
After BMOVE
1 to 199
200 to 599
600 to 800
801 to 1000
1 to 199
200 to 400
401 to 800
801 to 1000

First all pointers are stored in a dummy array with Bmove. The text, or even better, the text pointer, is moved between the end of the block and the cursor position at the beginning of the block. Finally, the stored block pointers are brought to the new position. All this happens without touching the line storage area.

If you take a look at the following illustration, you will see that no changes are made in the area between the beginning of the block and the cursor position. The complete operation is a swap of block pointer area and the area between end of the block pointer area and the cursor position.



For the second case where the block follows the cursor line, the block pointer area is stored, then the area between cursor position and beginning of the block is moved up. The block pointers fit into the now free positions. Finally, the block pointers are inserted again.

This procedure can probably not be optimized any further, all you can do is combine a few lines. But be careful that clarity does not suffer.

By the way, the total operations for block moving are only supposed to occur under two conditions. First: the cursor is not in the block. Second: the cursor should not follow the end of the text, because between the block and the end of text, two empty lines would be inserted and change the length of the text. This condition is not really necessary, but you have to change the algorithm if you want to eliminate it.

You also must display all the changes on the screen and the new position of the block has to be marked.

```
IF command_code%=87  ! Shift+F4: Move Block
 IF block!
  '
  ' Cursor can not be in the block.
  '
  IF upper%+CRSLIN<block_start% OR
  upper%+CRSLIN>block_end%
   '
   in_line%=upper%+CRSLIN
   '
   ' Only if there are no empty lines between
   ' the destination position and the
   ' end of the text.
   '
   IF in_line%<=number_lines%+1
    blocksize%=block_end%-block_start%+1
    '
    ' Put the old block pointers into a
    ' dummy array.
    '
    dest_addr%=base%+(block_start%-1)*4
```

```
source_addr%=dummy_base%
number%=blocksize%*4
IF number%>0
 BMOVE dest_addr%,source_addr%,number%
ENDIF
'
' If the defined block is in front
' of the cursor:
'
IF block_end%<upper%+CRSLIN
 '
 ' Move area between the block and
 ' the cursor down.
 '
 dest_addr%=base%+block_end%*4
 source_addr%=base%+(block_start%-1)*4
 number%=(upper%+CRSLIN-block_end%-1)*4
 IF number%>0
  BMOVE dest_addr%,source_addr%,number%
 ENDIF
 '
 ' Set the block pointers at
 ' the new position.
 '
 dest_addr%=dummy_base%
 source_addr%=base%+(block_start%-
 1+(upper%+CRSLIN-block_end%)-1)*4
 number%=blocksize%*4
 IF number%>0
  BMOVE dest_addr%,source_addr%,number%
 ENDIF
 '
 ' Adjust block markings
 '
 block_start%=block_start%-1+upper%+CRSLIN-
 block_end%
 block_end%=block_start%+blocksize%-1
ENDIF
```

```
  '
  ' If block follows the cursor:
  '
  IF block_start%>upper%+CRSLIN
   '
   ' Move the area between the block
   ' and the cursor up.
   '
   dest_addr%=base%+(upper%+CRSLIN-1)*4
   source_addr%=dest_addr%+blocksize%*4
   number%=(block_start%-upper%-CRSLIN)*4
   IF number%
    BMOVE dest_addr%,source_addr%,number%
   ENDIF
   '
   ' Set block pointers at new position.
   '
   dest_addr%=dummy_base%
   source_addr%=base%+(upper%+CRSLIN-1)*4
   number%=blocksize%*4
   IF number%>0
    BMOVE dest_addr%,source_addr%,number%
   ENDIF
   '
   ' Adjust block pointers.
   '
   block_start%=upper%+CRSLIN
   block_end%=block_start%+blocksize%-1
  ENDIF
  '
  upper%=block_start%-1
  @output_from_line(upper%+1)
  PRINT AT(1,1);
 ELSE
  PRINT CHR$(7);       ! Error bell
 ENDIF
ELSE
```

```
    ALERT 1," |Cursor is in the block.",1,"
    Return ",a%
   ENDIF
  ENDIF
ENDIF
```

3.9.4 Copy Block

The algorithm to copy a block must change the lines as stored in memory. The lines of the block must be doubled. Copies of block lines are put at the end of the storage area. Then the pointers from the position where the block copy is supposed to go will be moved up by one block length to make room for the block copy. Then we must point the pointers to this copy.

As usual, a few points have to be considered. The maximum allowed number of lines can not be exceeded. Also, the copy can not cause additional empty lines to be inserted. Of course, you can remove these conditions and rewrite the algorithm. In addition, if the copy goes before the original, the block borders must be changed. The whole routine will then look like this:

```
IF command_code%=86  ! Shift+F3: Block Copy
 IF block!
  dest_line%=upper%+CRSLIN
  '
  ' Only when there are no empty lines
  ' between the destination position and
  ' the end of the text.
  '
  IF dest_line%<=number_lines%+1
   blocksize%=block_end%-block_start%+1
   '
   ' Only if the maximum number of lines is
   ' not exceeded.
   '
   IF number_lines%+blocksize%<=max_lines%
    '
```

```
' Double lines which will be copied
' into storage.
'
pos_original%=block_start%
pos_copy%=number_lines%+1
FOR cnt%=1 TO blocksize%
 line$(pos_copy%)=line$(pointer%(
 pos_original%))
 INC pos_original%
 INC pos_copy%
NEXT cnt%
'
' Move block pointer
'
dest_addr%=base%+(dest_line%-1)*4
source_addr%=dest_addr%+blocksize%*4
number%=(number_lines%-dest_line%+1)*4
IF number%>0
 BMOVE dest_addr%,source_addr%,number%
ENDIF
'
' Point at copy:
'
point_at%=number_lines%+1
FOR cnt%=dest_line% TO
(dest_line%+blocksize%-1)
 pointer%(cnt%)=point_at%
 INC point_at%
NEXT cnt%
'
' If the copy is before the original.
' block markings must be corrected.
'
IF block_start%>dest_line%
 ADD block_start%,blocksize%
 ADD block_end%,blocksize%
ENDIF
'
ADD number_lines%,blocksize%
@output_from_line(upper%+1)
```

```
    ELSE
     ALERT 1,"Not enough space in memory|for
     this operation.",1," Return ",a%
    ENDIF
   ELSE
    PRINT CHR$(7); ! Error bell
   ENDIF
  ENDIF
ENDIF
```

3.9.5 Block Merging

Block merging means that data is loaded from the disk, and inserted at the cursor position. The algorithm is a kind of cross between Load Text and Copy Block. The same loading algorithm can be used as with Load Text; and the only difference between this and Copy Block is that the lines to be inserted are not coming from a different text position, but from the disk.

Since we selected the shorter and slower procedure to load text with LINE INPUT, the whole long listing of the loading algorithm does not have to be rewritten. If this seems to be too slow for you, just replace it with the faster INPUT$ procedure.

```
IF command_code%=89  ! Shift+F6: Block Merge
 '
 ' Merge Block is basically the same
 ' as Copy Block.
 '
 ' Store cursor position
 '
 char_pos%=CRSLIN
 ' Hide cursor
 PRINT CHR$(27)+"f";
```

```
'
' Call Fileselector with title
'
CLS
BOX 157,29,482,53
BOX 159,31,480,51
TEXT 271,47,"Merge Block"
FILESELECT path$,"",filename$
@store_path
'
' Load the block line by line, if file
' was selected, and Cancel Button not pressed.
'
IF NOT (RIGHT$(filename$)="\" OR filename$="")
 dest_load%=number_lines%+1
 OPEN "I",#1,filename$
 REPEAT
  LINE INPUT #1,line$(dest_load%)
  INC dest_load%
 UNTIL EOF(#1) OR dest_load%=max_lines%
 CLOSE #1
 blocksize%=dest_load%-number_lines%-1
 '
 IF dest_load%=max_lines%
  ALERT 1," |This file is too large.",1,"
  Return ",a%
 ELSE
  '
  IF blocksize%>0
   dest_line%=upper%+char_pos%
   '
   ' Move block pointer
   '
   dest_addr%=base%+(dest_line%-1)*4
   source_addr%=dest_addr%+blocksize%*4
   number%=(number_lines%-dest_line%+1)*4
   IF number%>0
    BMOVE dest_addr%,source_addr%,number%
   ENDIF
```

```
     '
     ' Point to copy
     '
     point_at%=number_lines%+1
     FOR cnt%=dest_line% TO
     (dest_line%+blocksize%-1)
      pointer%(cnt%)=point_at%
      INC point_at%
     NEXT cnt%
     '
     ' Set block markings
     '
     block!=TRUE
     block_start%=dest_line%
     block_end%=block_start%+blocksize%-1
     upper%=block_start%-1
     ADD number_lines%,blocksize%
   ENDIF
  ENDIF
 ENDIF
 '
 ' Show cursor and redraw screen
 PRINT CHR$(27)+"e";
 @output_from_line(upper%+1)
 PRINT AT(1,1);
ENDIF
```

3.9.6 Storing and Printing of Block

Printing and storing a block are two functions which are very similar. One functions sends the lines to the disk, the other to the printer. Because of the simplicity of the functions it is not necessary to write much commentary.

```
IF command_code%=88  ! Shift+F5: Store
 IF block!
```

```
'
' Store cursor position
char_pos%=CRSLIN
'
' Hide cursor
'
column%=CRSCOL
PRINT CHR$(27)+"f";
'
' Fileselector with title
'
CLS
BOX 157,29,482,53
BOX 159,31,480,51
TEXT 263,47,"Save Block"
FILESELECT path$,"",filename$
@store_path
'
' If valid filename selected, and Cancel
' button was not selected, store the block
' line by line.
'
IF NOT (RIGHT$(filename$)="\" OR
filename$="")
 OPEN "O",#1,filename$
 FOR cnt%=block_start% TO block_end%
  PRINT #1,line$(pointer%(cnt%))
 NEXT cnt%
 CLOSE #1
ENDIF
'
' Show cursor and restore screen
'
CLS
PRINT CHR$(27)+"e"
@output_from_line(upper%+1)
PRINT AT(column%,char_pos%);
```

```
  ELSE
    ALERT 1," |No block has been defined!",1,"
    Return ",a%
  ENDIF
ENDIF
```

Block Print is even easier and uses the procedure *to_print(block_start%,block_end%)*:

```
IF command_code%=90   ! Shift+F7: Block Print
  IF block!
    ALERT 2,"Send the defined block|to an
    attached printer?",1," Yes | No ",a%
    IF a%=1
      @to_printer(block_start%,block_end%)
    ENDIF
  ENDIF
ENDIF
```

With that, all necessary block functions have been discussed. The function key Shift + F9 is still free, perhaps you can think of another useful block function to include here.

3.10 Additional Helpful Functions

Only a few small easily implimented functions remain. These functions are also very useful.

3.10.1 Jumping the Cursor in a Line

Jumping a cursor in a line means, that the cursor can be moved within a line in larger steps than just the single steps available by pressing the cursor key. The <Control> + the Cursor right key moves the cursor to the right by five characters and <Control> + the Cursor left moves the cursor to the left by five characters. Both listings are part of procedure *command_2*.

```
IF command_code%=116  ! Control+Cursor right
 ' Move cursor five spaces right
 IF CRSCOL<75
  PRINT AT(CRSCOL+5,CRSLIN);
 ELSE
  PRINT AT(80,CRSLIN);
 ENDIF
ENDIF
'
'
'
IF command_code%=115  ! Control+Cursor left
 ' Move cursor five spaces left
 IF CRSCOL>5
  PRINT AT(CRSCOL-5,CRSLIN);
 ELSE
  PRINT AT(1,CRSLIN);
 ENDIF
ENDIF
```

3.10.2 Cursor Position Announcement

We do not have a line on the screen, which gives the actual cursor position. One advantage of this is that scrolling is much faster. But some method is needed to tell the user at what point in the text field he is working.

By pressing <Control> + <W> ('W' stands for where) an alert box is displayed with this information. This method is somewhat awkward, but better than none at all. It is part of the procedure *command_1*.

```
IF command_code%=23
 ' Control+W: Actual Cursor Position.
 ALERT 1,"Actual Cursor Position:|Line:
 "+STR$(upper%+CRSLIN)+"|Column:
 "+STR$(CRSCOL),1," Return ",a%
ENDIF
```

3.10.3 Go To Line

We also should have some method of jumping to any line within the text field. The following routine is called by pressing <Control> + <G> ('G' stands for goto). It belongs in procedure command_1.

```
IF command_code%=7   ! Control+G
 ' Hide Cursor
 PRINT CHR$(27)+"f";
 '
 GET 0,0,639,20,top_scrn$
 char_pos%=CRSLIN
 column%=CRSCOL
 '
 PRINT AT(1,1);SPACE$(80);
 LINE 0,18,639,18
 LINE 0,19,639,19
 PRINT AT(1,1);"Goto line#: "
 PRINT AT(16,1);
 FORM INPUT 5 AS jump$
 '
 ' Show cursor and restore screen
 '
 PUT 0,0,top_scrn$
 PRINT CHR$(27)+"e";
 '
 IF VAL(jump$)<1 OR VAL(jump$)>max_lines%
  alrt$="There is no line# "+jump$
  ALERT 1,alrt$,1," Return ",a%
 ELSE
  upper%=VAL(jump$)-1
  @output_from_line(upper%+1)
  PRINT AT(1,1);
 ENDIF
 '
ENDIF
```

3.10.4 Go to Start of Text, End of Text, Top of Screen Page

Jumps to a few marked points of the text should also be possible. If you'd like, you can in addition build in jumps to block begin and block end. The jump to the end of the text with <Control> + <Z> belongs in procedure *command_1*, the functions for jumping to the start of the text and the top of the screen page belong in procedure *command_2*.

```
IF command_code%=26
 ' Control + Z : Goto the end of the text
 upper%=MAX(1,number_lines%-25)
 @output_from_line(upper%+1)
ENDIF
```

```
IF command_code%=71  !Home Key
 PRINT AT(1,1);
ENDIF
```

```
IF command_code%=119   ! Control+Home
 IF upper%>0
  upper%=0
  @output_from_line(1)
  PRINT AT(1,1);
 ENDIF
ENDIF
```

3.10.5 Adjusting the Cursor Speed

If you snoop around a little bit in the ST's bag of tricks, you'll find a function with which several cursor parameters can be adjusted. This function is XBIOS(35,av,ge). The *av* stands for respond delay. This is the time that passes between the first reaction to a keypress and the start of the repeat function of holding a key down.

The variable *ge* is the speed of the cursor, which can be adjusted in the following manner (Listing in procedure *command_1):*

```
IF command_code%=20
 ' Control + T: Set cursor speed
 ALERT 2,"Which cursor speed|would you
 like?",2,"Fast|Medium|Slow",a%
 IF a%=3
  INC a%
 ENDIF
 VOID XBIOS(35,10,a%)
ENDIF
```

3.11 Perspectives

Our editor is finally complete! As with any other program, many additional features can be added. You might decide to turn this simple text editor into a elaborate Word Processor.

3.11.1 Additional Functions

Additional functions now depends only on your imagination and needs. The following points are just a few examples of many.

From the explanation provided, you should be able to add any features you might want. For example, sometimes it is very handy to be able to cut out parts of a text from one documentand just add them into another one. It has to be possible, therefore, to work on two text files at the same time. No windows are needed, just change a command to switch from one document to another. To achieve this, you will need two line and pointer arrays for swapping of text portions and memory. All this can be easily done.

Several available functions could be extended, i.e. searching with ignoring capitalization of letters, replacing without checking in the whole text, etc. To be able to save text when no free disk is at hand, an option to delete files on the disk can be built in.

The insertion of these commands happens the same way as all the other built in ones. You can see that the path for writing a deluxe editor is not blocked.

3.11.2 Towards Text Processing

You have probably asked yourself several times already, how difficult would it be to make a complete Word Processor out of this editor. What is still missing? The most important functions, which our editor does not have, would be block-set, word wrap, and the ability to use different kinds of fonts.

Additional needed functions are the adjusting of left and right margins, justification, a small footnote management function, page format management, etc. More difficult things can be left out. If you plan such an extension, you should add a pull down menubar.

Experiment and enjoy!

Index

Index

cut here

Concepts In Programming Accessories Order Form

Name ____________________

Address ____________________

City ____________ State ______ Zip ________

Phone (____) ____________

Qty		Amount
____	Concepts In Programming Program Disk @ $12.95	____
____	GFA BASIC Reference Card @ $3.95	____
	Shipping	$ 3.00
	Total	$ ____

Select a method of payment from the following:

□ Check or Money Order (Payable to MichTron Inc.)

□ Please charge my: □ Visa □ Mastercard

Card No. ____________________ Exp. Date ______

Signature ____________________

Send Orders To:
MichTron Inc.
576 Telegraph
Pontiac, MI 48053

MichTron®

Come and join us at the Roundtable,TM Where the GEnieTM and the Griffin meet!

Does this sound like a fantasy? Well, it may just be a dream come true! When General Electric's high-tech communications network meets MICHTRON's programmers and support crew, ST users around the country will hear more, know more, and save more.

We know that our low prices and superior quality wouldn't mean as much to you without the proper support and service to back them up.

So we are now available on GEnie, the General Electric Network for Information Exchange. GEnie is a computer communications system which lets you use your personal computer, modem, and communication software to gain access to the latest news, product information, electronic mail, games, and MICHTRON's *own* Roundtable (See the special MICRODEAL Section for game information)!!

The Roundtable Special Interest Groups (SIG) gives you a means of conveniently obtaining news about our current products, new releases, and future plans. Messages directly from the authors give you valuable technical support of our products, and the chance to ask questions (usually answered within a single business day).

GEnie differs from other computer communication networks in its incredibly low fees. With GEnie, you don't pay any hidden charges or minimum fees. You pay only for the time you're actually on-line with the MICHTRON product support Roundtable, and the low first-time registration fee.

For more information on GEnie, follow this simple procedure for a free trial run. Then if you like, have ready your VISA, Mastercard or checking account number and you can set up your personal account immediately -- right on-line!

1. Set your modem for half duplex (local echo)--300 or 1200 baud.
2. Dial **1-800-638-8369.** When connected, type **HHH** and press **Return.**
3. At the U#= prompt, type **XJM11957,GENIE** and press **Return.**

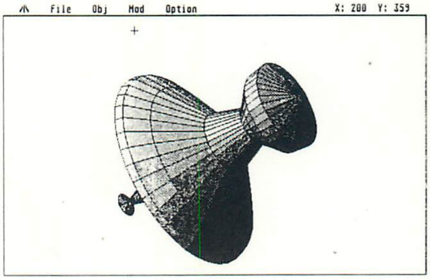
And don't forget, MICHTRON's Bulletin Board System, The Griffin BBS, is still going strong (the griffin is the half-lion/half-eagle creature on our logo). Our system is located at MICHTRON headquarters in Pontiac, Michigan. For a trial run, call (313) 332-5452.

GEnie and Roundtable are Trademarks of General Electric Information Services.

Concepts In Programming

Gottfried P. Engels guides you through the mystery of programming advanced applications in **GFA BASIC**, providing helpful tips for improving your programs. Simple techniques illustrate the power you'll unleash with **GFA BASIC**. All programs and examples will work with any version of **GFA BASIC**.

This book is ideal for the programmer who has written a number of small programs and now feels ready to attack the advanced concepts involved in creating larger, more intricate applications.



Example programs include an *advanced* 3D object editor and a text editor. The reader will be walked through the creation of both these invaluable programs, plus given hints on how to customize them for personal use. When completed, you will not only have a wealth of programming skills, but also two indispensable software applications as well.

Though intended for the Intermediate programmer, **GFA BASIC** users at *all* levels of ability will profit from *Concepts in Programming*.

For all Atari ST computers

Requires GFA BASIC 2.0 or GFA BASIC 3.0

0 09445 00153 5